1994

Data Partition and Migration for High Performance Computation in Distributed Memory Multiprocessors

Nagesh Anupindi
University of Rhode Island

DATA PARTITION AND MIGRATION FOR HIGH PERFORMANCE COMPUTATION IN DISTRIBUTED MEMORY MULTIPROCESSORS

BY
NAGESH ANUPINDI

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN
ELECTRICAL AND COMPUTER ENGINEERING
MAJOR PROFESSOR: QING YANG

THE UNIVERSITY OF RHODE ISLAND
1994

DOCTOR OF PHILOSOPHY DISSERTATION

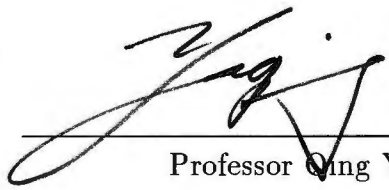
OF

NAGESH ANUPINDI

APPROVED:

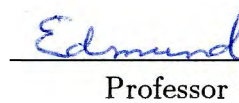
Dissertation Committee

Major Professor

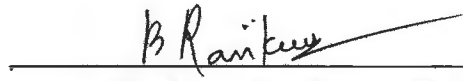


Professor Qing Yang

Professor Ramdas Kumaresan



Professor Edmund Lamagna



Professor Ravikumar Bala

DEAN OF THE GRADUATE SCHOOL

THE UNIVERSITY OF RHODE ISLAND

1994

TO
MASTERS EK AND CVV,
AMMA, NAANNA, AND SUNDAR

Abstract

Data-partition and migration for efficient communication in distributed memory architectures are critical for performance of data parallel algorithms. This research presents a formal methodology for the process of data-distribution and redistribution using tensor products and stride permutations as mathematical tools. The algebraic expressions representing data-partition and migration directly operate on a data vector, and hence can be conveniently embedded into an algorithm. It is also shown that these expressions are useful for a clear understanding and to efficiently interleave problems that involve different data-distributions at different phases. This compatibility made us successfully utilize these expressions in developing and demonstrating matrix transpose and fast Fourier transform algorithms. Usage of these expressions for data interface generated efficient parallel implementation to solve Euler partial differential equation. An endeavor to minimize communication cost using expressions for data-distribution disclosed a routing scheme for Fourier transform evaluation. Results promised that for large parallel machines, this scheme is a solution to today's problems which feature enormous data. Finally, a unique data-distribution technique that effectively uses transpose algorithms for multiplication of two rectangular matrices is derived. Performance of these algorithms are evaluated by carrying out implementations on Intel's *i860* based *iPSC/860*, Touchstone Delta, Gamma, and Paragon supercomputers.

Acknowledgments

This is most pleasant part of writing for I get a chance to thank all those who helped make this dissertation an enjoyable one. I thank Professor James W. Cooley for inspiring the idea of Fourier transform algorithms with respect to parallel machines, for several helpful suggestions, and for recommending me for departmental financial assistance. I also thank Professor Qing Yang for providing financial support through NSF grant MIP-9208041, and for his valuable discussions and suggestions. He also introduced me to the area of high performance computing, and helped me to present this work in an elegant form. Special thanks to Professor Richard Tolimieri and Professor Myoung An for their valuable discussions, moral support, friendship, and financial support through Aware Inc., Cambridge, MA. I am also thankful to Prof. John Weiss at Aware Inc., for introducing and guiding me through the partial differential equation solvers.

I am grateful to the faculty/staff members and fellow students in the Department of Electrical Engineering, who never hesitated to offer timely assistance and warm friendship. I am also grateful to all members of my thesis committee, Professor Ramdas Kumaresan, Professor Edmund Lamagna, and Professor Jein-Chung Lo for their precious time and efforts. I would like to thank Professor Ravikumar Bala for being the chairman of the committee. I thank University of Rhode Island, and Intel Inc., for making their computer power available to carry out the necessary experiments in this dissertation without which this work is not feasible. Special thanks to Dr. Dane P. Kottke for his assistance.

I am indebted to my parents Mr. Kameswara Rao Anupindi and Mrs. Kameswari Anupindi for devoting their souls to give me this beautiful life, and my brother Mr. Sundar Ram Anupindi for memories of a life time.

Nagesh Anupindi

Preface

The demand for high speed computers has been more than existing computing power at any time in the computer era. Even very impressive electronic components could not satisfy today's thirst for performing enormous number of calculations involved in most of the practical applications. With these challenges, parallel processing is the way to achieve desired computing speeds. A parallel computer consists of a collection of processing units that assist together to solve an application. Architects of parallel computers have the freedom to select number of processing units, to link processors through various interconnections, to have shared or distributed memory, to design synchronous or asynchronous operations, etcetera.

For academic researchers, access to supercomputers is still limited. Nonetheless, usage of supercomputers by the community of scientists is increasing every year, and research projects performed on these became more ambitious and sophisticated. To solve problems once thought impractical, supercomputers have become friendly tools.

This dissertation addresses aspects in parallel systems which have distributed memory and feature independence from underlying interconnection network. The problems studied in this dissertation are based on mathematical tools which can represent algorithms for parallel systems. Examples are used as often as possible to illustrate these tools. Distributing the problem onto processors is modeled using these tools while they were proven to be helpful to optimize old solutions as well as to derive new solutions. A list of references to publications where related problems and algorithms are treated is provided at the end.

Nagesh Anupindi

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many scientific computations such as engineering, energy resource, medical, military, artificial intelligence, and basic research areas demand fast processing computers to achieve required computational performance. Without the existence of superpower computers, the study of many of these applications and the efforts to meet today's challenges could hardly be realized. Since device characteristics are approaching the physical limit, parallel processing is the only way to improve computing power further in order to meet its ever increasing demand. Research in cost-effective, high-speed, massively parallel, and reliable supercomputers has become a very active field in computer engineering.

Two distinct and important parallel computer architectures are shared-memory, and message-passing systems [1, 2]. A *shared-memory* machine has a single global memory accessible to all processors such as IBM RP3, Encore Multimax, Cray X-MP, and many workstations. Its important feature is that communication between nodes is done by reading from and writing into the shared-memory. However, the shared-memory builds a barrier for increasing number of processing elements. *Message-passing* systems, also known as distributed memory system, allocate a stipulated amount of memory to each processing element but data does not form a single address space [3]. Communication among processors is done through message-passing. Intel *i*PSC, nCube, and Caltech Mark II hypercube belong to this category. In this

architecture, if an application shares data at distinct nodes, a programmer specifically commands to port data from one node to another. Since no resources such as data, cache, CPU time, etc., are globally shared besides the links (a link is accessible to a very limited number of processors), message-passing systems are scalable and preferred by researchers for solving larger problems.

One important characteristic of message-passing machines (also known as distributed memory systems) is that there is a significant timing difference between local and remote data accesses. Remote data access involves message-passing among processors. This message-passing process takes a significant amount of total execution time of a computational procedure. The amount of remote data accesses needed to accomplish a given computation mainly depends upon how data are initially allocated to processors. We refer to this initial data allocation as *data-partition*. Efficient data-partition in distributed memory systems is essential for achieving high performance of data parallel programs. An optimal data-partition for one individual algorithm (computation module) may not be optimal for others. Therefore, optimized data-partition for applications that involve a number of computation algorithms or modules present an interesting challenge to parallel processing researchers.

Optimization of data-partition is achieved mainly by constraining to a rule that number of message-passings should be minimum for a given system architecture. The system architecture of a distributed memory system is mainly determined by its interconnection structure. Various interconnection networks and evaluation of their performance have been reported in the literature [4, 5, 6, 7]. Some well known network schemes for message-passing systems are ring, tree, mesh, hypercube and star connections. Processors that are directly connected are called neighbors. Processors that are not connected through a direct link are called non-neighbors. Distance between processors is calculated based on the minimum number of processors through which they are connected. It was also observed that the farther the node is, the longer it takes to port data. Hence, the performance of an implementation can be optimized by minimizing the distance a node has to travel to get data. Extensive

efforts have been put forth in minimizing communication distance among processors in a research field called *parallel programming* [8, 9].

Recently, a very important technological advancement has occurred in interconnection networks to minimize communication cost known as *wormhole routing* [10, 11]. In this scheme, a message is divided into a number of *flits* (flow control digits). Once the header flit of a message acquires a channel, it governs the route of that message and the remaining flits follow in a pipelined fashion. This pipelined nature of wormhole routing procedures resulted in a network latency that is relatively insensitive to the distance between processors [12], which in turn has made communication cost to predominantly depend upon message lengths and the number of messages. These features have attracted commercial multicomputers such as the Intel's Touchstone Delta and Intel's Paragon which use wormhole routing in a 2D mesh, and MIT's J-machine which uses wormhole routing in 3D mesh. Ametek 2010 used a mesh network renamed as Symult 2010 after adopting wormhole routing. The nCube-2 which originally used hypercube topology has now adopted wormhole routing.

With wormhole routing, passing a message from one node to another requires a specific start-up (overhead) time, t_{start}, and a transmission time that depends upon the length of the message. Let $t_{element}$ be the time for an 8-byte data to pass through an acquired channel. Then a simple model for total communication time for a message of l-bytes is given by

$$t_{total} = t_{start} + (l/8)\, t_{element}. \tag{1.1}$$

We conducted experiments to determine the relationship between start-up time and transmission time on several machines. Table 1.1 shows the results of our experiments to obtain start-up time and time to communicate 8-bytes (one complex number) of data on Intel's Paragon, Gamma, and Touchstone Delta. Results are obtained by averaging observed timings over 100 samples for passing a message from each node to every other node. Possible machine-partitions (machine-partition is a cluster of nodes that is subset of all the nodes on a machine. For example, a cluster of 16

nodes on 512-node Touchstone Delta is a 16-node machine-partition) are considered for each machine to observe the effect of node's distance on total communication time. It is observed that the ratio between start-up time and transmission time remains almost constant irrespective of physical distance between nodes, implying performance results close to a fully connected distributed system. Understanding such features of new machines with respect to their latency of message-passing helps parallel programmers to develop efficient algorithms.

Because of the importance of data-partition in data parallel programs, we focused our attention on the study of partitioning an application and migrating the necessary data among processors. This dissertation formulates the data-partitioning schemes and derives variants of migrating schemes in distributed memory multiprocessor systems using tensor algebra. The essential feature of this formulation is that data-partition and migration are represented using simple tensor algebraic expressions. Therefore, they can form parts of an algorithm that is already written in tensor algebraic notation. Furthermore, by using tensor notation and stride permutations, our formulation is simple and compact without having to deal with complicated indices in complex data structures. Such a clear mathematical representation of storage schemes helps parallel programmers greatly to look into inherent structure(s) of an algorithm and the associated communication cost.

With the newly proposed formulations, optimal data-migration at interfaces between computation modules become straightforward algebraic manipulations. This research demonstrates the manipulations of fast Fourier transform (FFT) algorithms for efficient implementations. These algorithms are applied to an application that solves Euler partial differential equation using wavelet-Galerkin method and achieved a significant improvement in the overall performance. Then, we have designed an efficient two-dimensional FFT algorithm for distributed systems using algebraic expressions for mesh-division data-partitioning. This design is shown to be a solution to the problems featuring huge data size, large machines and higher dimensionality. Optimal data-partition is considered also for matrix multiplication algorithms

Number of Nodes	Paragon		Gamma		Delta	
	t_{start} (msec)	$t_{element}$ (μsec)	t_{start} (msec)	$t_{element}$ (μsec)	t_{start} (msec)	$t_{element}$ (μsec)
4	4.330	0.922	2.753	5.734	1.046	2.560
8	4.411	0.939	2.981	6.357	1.355	3.328
16	4.445	0.934			1.548	3.757
32	4.503	0.985			1.777	4.541
64	4.512	0.989			1.831	4.521
128					1.910	4.658
256					1.921	4.708

Table 1.1: Results of experiments to determine the start-up and transmission times

where we have shown that distributed transpose algorithms can be efficiently used for multiplying two rectangular arrays.

In order to demonstrate the usefulness and significance of our data-partition expressions, we carried out implementations of our algorithms and applications on Intel's supercomputers: *i*PSC/*860*, Paragon, Gamma, and Touchstone Delta.

This dissertation is organized as follows: Chapter 2 reviews the tensor algebraic notation, and related theorems that are required for a better understanding of the concepts in our contributions. It also presents a survey of the existing literature that is related to this dissertation. Chapter 3 presents definitions for three data-allocation schemes in tensor algebraic notation. Demonstration of these expressions is presented for the case of matrix transpose algorithms using all the three distinct data-allocation schemes. This chapter also presents tensor product formulation of two-dimensional discrete Fourier transform (2D-DFT) for row-division data-partition using the transpose algorithms. Computation of vorticity and stream functions in two-dimensional fluid turbulence using 2D-DFT is carried out in Chapter 4 to demonstrate the usage of data-allocation expressions to efficiently interface two computation modules

that are efficient for two different data-partition schemes. A new and highly efficient approach to evaluate large 2D-DFTs using large parallel computers is derived using tensor algebra in Chapter 5. Chapter 6 considers the matrix multiplication algorithms for distributed memory systems via matrix transpose algorithm by presenting a unique data allocation scheme for rectangular multiplicands. Chapter 7 discusses future research and possible extensions of the methods developed in this work to other algorithms.

Chapter 2

Preliminaries and Related Work

2.1 Introduction

In a distributed environment, implementation procedure for most of the applications involves dividing the main computational task into (a) *local tasks* that depend upon data residing at a node's local memory, and (b) *global tasks* that depend upon data residing at more than one node. Such an identification and separation gives an estimation of the degree of node balance in an implementation and also the inherent message-passing overheads. Tensor algebra is a mathematical language that aids to identify, express, and analyze these tasks in an algorithm. Two most important operations in tensor algebra are tensor products, and stride permutations. In the following sections, we introduce these operations and demonstrate their importance with respect to parallel machines. This notation is used in Chapter 3 to develop a set of formal definitions for data-partition schemes. Later parts of Chapter 3 use the same notation to express matrix transpose algorithms, and multidimensional fast Fourier transform (FFT) calculations with an emphasis on their implementation aspects for a distributed memory system. Also, Chapters 4 and 5 deal with variants of FFT algorithms using the same notation.

Sections 2.2 through 2.5 review the necessary notation and relevant theorems to this dissertation from tensor algebra. Survey of the literature related to data-partitioning schemes is presented in Section 2.6 while that for FFT algorithms and matrix computations are presented in Sections 2.7 and 2.8, respectively. Section 2.9 gives a brief description of the machines on which experiments in this dissertation are conducted. Section 2.10 concludes the chapter.

2.2 Operators *Mat* and *Vect*

Let $\mathbf{x}$ be KJ-element vector: $[x_0\ x_1\ \ldots,\ x_{KJ}]^T$. The matrix operator, $Mat_{K\times J}$, converts $\mathbf{x}$ into a $K \times J$ matrix as follows.

$$\mathbf{X} = Mat_{K\times J}(\mathbf{x}) = \begin{bmatrix} x_0 & x_K & \cdots & x_{(J-1)K} \\ x_1 & x_{K+1} & \cdots & x_{(J-1)K+1} \\ x_2 & x_{K+2} & \cdots & x_{(J-1)K+2} \\ \vdots & \vdots & \ddots & \vdots \\ x_{K-1} & x_{2K-1} & \cdots & x_{JK-1} \end{bmatrix} \tag{2.2}$$

The inverse operation of *Mat*, $Vect_{KJ}$, forms a linear array according to column-major scheme as follows.

$$\mathbf{x} = Vect_{KJ}(\mathbf{X}) = Vect_{JK}\left(\begin{bmatrix} x_0 & x_K & \cdots & x_{(J-1)K} \\ x_1 & x_{K+1} & \cdots & x_{(J-1)K+1} \\ x_2 & x_{K+2} & \cdots & x_{(J-1)K+2} \\ \vdots & \vdots & \ddots & \vdots \\ x_{K-1} & x_{2K-1} & \cdots & x_{JK-1} \end{bmatrix}\right) \tag{2.3}$$

2.3 Stride Permutation

Stride permutations are natural way of representing data-shuffling operations. We use $P(L_x, S)$ to represent a stride permutation operation on a vector of length L_x with stride S. Let $\mathbf{x}$ be an L_sS-element vector and $L_x = L_sS$. Then, the stride

permutation, $\mathbf{y} = P(L_x, S)\mathbf{x}$, performs the following operations. The first L_s elements of $\mathbf{y}$ are obtained by picking up elements of $\mathbf{x}$ starting at x_0 and then each Sth element of $\mathbf{x}$: that is, $\{x_0, x_S, \ldots x_{(L_s-1)S}\}$. The next L_s elements of $\mathbf{y}$ are obtained in the same way starting at x_1 of $\mathbf{x}$: $\{x_1, x_{S+1}, \ldots, x_{(L_s-1)S+1}\}$, and so on. Therefore, the stride permutation operation, $P(L_x, S)$, is an $L_x \times L_x$ size matrix that is filled with zeros and ones.

Example 2.1 *Permutation matrix* $P(6,3)$ *shown below is operating on vector* $\mathbf{x} = [x_0\ x_1\ x_2\ x_3\ x_4\ x_5]^T$, *and denoted as* $\mathbf{y} = P(6,3)\mathbf{x}$.

$$\mathbf{y} = \begin{bmatrix} x_0 \\ x_3 \\ x_1 \\ x_4 \\ x_2 \\ x_5 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}}_{P(6,3)} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \tag{2.4}$$

2.4 Tensor Product

Tensor product is a binary operator between two matrices of any size. Given two matrices $\mathbf{A}$ and $\mathbf{B}$ of sizes $M_A \times N_A$ and $M_B \times N_B$, respectively, a new matrix, $\mathbf{C}$, dimensioned $M_A M_B \times N_A N_B$ can be generated by tensor product of $\mathbf{A}$ and $\mathbf{B}$ as:

$$\mathbf{C} = \mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{(0,0)}\mathbf{B} & a_{(0,1)}\mathbf{B} & a_{(0,2)}\mathbf{B} & \cdots & a_{(0,N_A-1)}\mathbf{B} \\ a_{(1,0)}\mathbf{B} & a_{(1,1)}\mathbf{B} & a_{(1,2)}\mathbf{B} & \cdots & a_{(1,N_A-1)}\mathbf{B} \\ a_{(2,0)}\mathbf{B} & a_{(2,1)}\mathbf{B} & a_{(2,2)}\mathbf{B} & \cdots & a_{(2,N_A-1)}\mathbf{B} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{(M_A-1,0)}\mathbf{B} & a_{(M_A-1,1)}\mathbf{B} & a_{(M_A-1,2)}\mathbf{B} & \cdots & a_{(M_A-1,N_A-1)}\mathbf{B} \end{bmatrix} \tag{2.5}$$

where $a_{(i,j)}$ is the element at ith row and jth column of $\mathbf{A}$, and $a_{(i,j)}\mathbf{B}$ is scalar matrix multiplication. In other words, tensor product is a list of all possible combinations of multiplications of one matrix's elements with the other's.

Example 2.2 *Consider two matrices* $\mathbf{A}$ *and* $\mathbf{B}$ *as:*

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad \textit{and} \quad \mathbf{B} = \begin{bmatrix} 10 & 11 & 12 \\ 13 & 14 & 15 \end{bmatrix}.$$

Then

$$\mathbf{C} = \mathbf{A} \otimes \mathbf{B} = \left[\begin{array}{c|c} \mathbf{B} & 2\mathbf{B} \\ \hline 3\mathbf{B} & 4\mathbf{B} \end{array}\right] = \left[\begin{array}{ccc|ccc} 10 & 11 & 12 & 20 & 22 & 24 \\ 13 & 14 & 15 & 26 & 28 & 30 \\ \hline 30 & 33 & 36 & 40 & 44 & 48 \\ 39 & 42 & 45 & 52 & 56 & 60 \end{array}\right]$$

according to equation (2.5).

Two types of tensor products are of special interest to us here from the point of parallel computations. One has an identity matrix on the left-hand side of the tensor product as $\mathbf{I} \otimes \mathbf{A}$, called *prior identity matrix*, and the other has an identity matrix on the right-hand side such as $\mathbf{A} \times \mathbf{I}$, referred as *post identity matrix*. For the rest of the discussion in this section, let a vector $\mathbf{x}$ be of length $L_x = JN_A$, vector $\mathbf{y}$ be of length $L_y = JM_A$, matrix $\mathbf{I}_J$ be identity matrix of size $J \times J$, and $\emptyset_{M_A \times N_A}$ be a null matrix of size $M_A \times N_A$.

When tensor product of an identity matrix $\mathbf{I}_J$ with a matrix $\mathbf{A}$ of size $M_A \times N_A$ is applied on a vector $\mathbf{x}$, it can be written as

$$\mathbf{y}_{JM_A} = [\mathbf{I}_J \otimes \mathbf{A}_{M_A \times N_A}]\, \mathbf{x}_{JN_A} \tag{2.6}$$

The above equation can be expanded using the definition of tensor product as

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{L_y-1} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{M_A \times N_A} & \emptyset_{M_A \times N_A} & \emptyset_{M_A \times N_A} & \cdots & \emptyset_{M_A \times N_A} \\ \emptyset_{M_A \times N_A} & \mathbf{A}_{M_A \times N_A} & \emptyset_{M_A \times N_A} & \cdots & \emptyset_{M_A \times N_A} \\ \emptyset_{M_A \times N_A} & \emptyset_{M_A \times N_A} & \mathbf{A}_{M_A \times N_A} & \cdots & \emptyset_{M_A \times N_A} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \emptyset_{M_A \times N_A} & \emptyset_{M_A \times N_A} & \emptyset_{M_A \times N_A} & \cdots & \mathbf{A}_{M_A \times N_A} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{L_x-1} \end{bmatrix}, \tag{2.7}$$

which can also be expressed using operators Mat and $Vect$ as

$$\mathbf{y}_{JM_A} = Vect_{M_A J}\{\mathbf{A}_{M_A \times N_A}(Mat_{N_A \times J}(\mathbf{x}_{L_x}))\} \tag{2.8}$$

On a J-processor architecture, this representation gives a mechanism of simultaneously operating matrix $\mathbf{A}$ on different parts of input data by different processors.

Example 2.3 *Consider a 4-processor machine and the following operational matrix* $\mathbf{A}$ *to be operated on vector* $\mathbf{x}$*:*

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad \textit{and} \quad \mathbf{x} = [x_0\ x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7]^T .$$

Then, $\mathbf{y} = (\mathbf{I}_4 \otimes \mathbf{A})\ \mathbf{x} =$

$$\left[\begin{array}{c} x_0 + x_1 \\ x_0 - x_1 \\ \hline x_2 + x_3 \\ x_2 - x_3 \\ \hline x_4 + x_5 \\ x_4 - x_5 \\ \hline x_6 + x_7 \\ x_6 - x_7 \end{array}\right] = \left[\begin{array}{cccccccc} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{array}\right] \left[\begin{array}{c} x_0 \\ x_1 \\ \hline x_2 \\ x_3 \\ \hline x_4 \\ x_5 \\ \hline x_6 \\ x_7 \end{array}\right]$$

Here, each processor executes one addition and one subtraction on a different part of $\mathbf{x}$*, where the node boundaries are represented by horizontal lines. Execution of the same task on 2-processor machine can be written as* $[\mathbf{I}_2 \otimes (\mathbf{I}_2 \otimes \mathbf{A})]$ *representing double amount of computation by each processor.*

Therefore, these tensor products with prior identity matrices can be used to determine the existence of local (parallel) tasks. In general, on a k-processor distributed memory machine, execution of $(\mathbf{I}_J \otimes \mathbf{A})$ would imply k local tasks, where $J = nk$ and n is a positive integer greater than zero.

If an identity matrix appears on the right-hand side of a tensor product, it is performed in a natural way for vector computers, that is, performing an operation

2.5 Some Useful Theorems

A number of properties that tensor products hold in combination with stride permutations will be useful in developing variants of a parallel algorithm. We will present these properties here without proof. Interested readers can find their proofs in [13, 14]. Similar to the notation in algebra of matrices, a complex tensor product formulation should be read from right to left.

Theorem 2.1 Associative Law: *Tensor product on a set of matrices is associative.*

$$(\mathbf{A}_{M_A \times N_A} \otimes \mathbf{B}_{M_B \times N_B}) \otimes \mathbf{C}_{M_C \times N_C} = \mathbf{A}_{M_A \times N_A} \otimes (\mathbf{B}_{M_B \times N_B} \otimes \mathbf{C}_{M_C \times N_C}) \quad (2.12)$$

Theorem 2.2 Distributive Laws: *When two matrices* $\mathbf{A}$ *and* $\mathbf{B}$ *are of the same size, following additive distribution law holds true irrespective of the size of matrix* $\mathbf{C}$.

$$\begin{aligned}(\mathbf{A}_{M \times N} + \mathbf{B}_{M \times N}) \otimes \mathbf{C}_{M_C \times N_C} \\ = \ (\mathbf{A}_{M \times N} \otimes \mathbf{C}_{M_C \times N_C}) + (\mathbf{B}_{M \times N} \otimes \mathbf{C}_{M_C \times N_C})\end{aligned} \quad (2.13)$$

Similarly, when two matrices $\mathbf{B}$ *and* $\mathbf{C}$ *are of the same size, following additive distribution law holds true irrespective of the size of matrix* $\mathbf{A}$.

$$\begin{aligned}\mathbf{A}_{M_A \times N_A} \otimes (\mathbf{B}_{M \times N} + \mathbf{C}_{M \times N}) \\ = \ (\mathbf{A}_{M_A \times N_A} \otimes \mathbf{B}_{M \times N}) + (\mathbf{A}_{M_A \times N_A} \otimes \mathbf{C}_{M \times N})\end{aligned} \quad (2.14)$$

Theorem 2.3 Multiplication of Tensor Products: *If* $N_X = M_A$ *and* $N_Y = M_B$, *then the following multiplication theorem holds true.*

$$\begin{aligned}(\mathbf{X}_{M_X \times N_X} \otimes \mathbf{Y}_{M_Y \times N_Y}) \ \ (\mathbf{A}_{M_A \times N_A} \otimes \mathbf{B}_{M_B \times N_B}) \\ = \ (\mathbf{X}_{M_X \times N_X} \ \mathbf{A}_{M_A \times N_A}) \ \otimes \ (\mathbf{Y}_{M_Y \times N_Y} \ \mathbf{B}_{M_B \times N_B})\end{aligned} \quad (2.15)$$

This theorem is quite often used to derive parallel or vector computations when identity matrices appear in the product.

Theorem 2.4 Commutative Law: *Interchanging the order of tensor product parameters results in permutations.*

$$(\mathbf{A}_{M_A \times N_A} \otimes \mathbf{B}_{M_B \times N_B}) = P(M_A M_B, M_A)\,(\mathbf{B}_{M_B \times N_B} \otimes \mathbf{A}_{M_A \times N_A})\,P(N_A N_B, N_B) \tag{2.16}$$

This theorem is quite useful in generating different communication structures of an algorithm.

Theorem 2.5 Inverse of Tensor Products: *Unlike the case in inverse of multiplication of two matrices, inverse of tensor product of two matrices does not change the order of its parameters.*

$$(\mathbf{A} \otimes \mathbf{B})^{-1} = \left(\mathbf{A}^{-1} \otimes \mathbf{B}^{-1}\right) \tag{2.17}$$

Theorem 2.6 Multiplication Theorem of Stride Permutations: *Any simple-stride permutation can be decomposed into two stride permutations when stride is a multiple of two integers.*

$$P(N_A N_B N_C, N_A N_B) = P(N_A N_B N_C, N_A) P(N_A N_B N_C, N_B) \tag{2.18}$$

Theorem 2.7 Parallel-Vector Tensor Factorization of Stride Permutations:

$$P(N_A N_B N_C, N_C) = [P(N_A N_C, N_C) \otimes \mathbf{I}_{N_B}]\,[\mathbf{I}_{N_A} \otimes P(N_B N_C, N_C)] \tag{2.19}$$

This is one of the very important theorems for implementation of a permutation on distributed memory systems to uncover extent of communication complexity hidden in that permutation. When parameter N_A is an integral multiple of number of processing elements, this theorem extracts local operations from operations that depend upon non-local data. A stride permutation can also be factorized in a different way leading to the following theorem:

Theorem 2.8 Vector-Parallel Tensor Factorization of Stride Permutations:

$$P(N_A N_B N_C, N_A N_B) = [\mathbf{I}_{N_A} \otimes P(N_B N_C, N_B)]\,[P(N_A N_C, N_A) \otimes \mathbf{I}_{N_B}] \tag{2.20}$$

Theorem 2.9 Inverse Stride Permutation:

$$P(N_A N_B, N_A)^{-1} = P(N_A N_B, N_B). \tag{2.21}$$

Theorem 2.10 Identity Stride Permutations:

$$P(N_A, N_A) = P(N_A, 1) = \mathbf{I}_{N_A} \tag{2.22}$$

2.6 Existing Data-Partition Representations

It is well known that data-distribution in distributed memory multiprocessors is essential to achieve high performance of data-parallel programs. Extensive research has been reported on data-decomposition optimization for distributed memory machines [15, 16, 17, 18, 19]. Research in this area can be crudely classified into two categories. One aims at finding optimal data-partitioning schemes for parallel loop constructs as part of compiler. It has been shown that the problem of finding an optimal data-partition is NP-complete [17, 20, 15]. Therefore, researchers have to rely on heuristic methods [20, 21, 22, 16, 23]. The other effort aims at special-purpose implementations and a large work force for developing optimal implementation of individual algorithms is reported [24, 25, 26].

Typically, an application requires a number of computation modules linked together to accomplish a specific computation. Global optimization depends not only on optimal implementation of the computational modules, but at least equally on the interface between these implementations as determined by the data partition and migration across processors.

In this dissertation, we present a systematic formulation for data-partition and migration on distributed memory multiprocessors in terms of tensor product notation and stride permutations [27]. Data-partition and migration are represented using simple tensor algebraic expressions highlighting the computational and communication complexity of parallel algorithms. Therefore, optimal data-partition and

migration at interfaces between different algorithms becomes straightforward tensor algebraic manipulations with the aid of well-established theorems in this field. Furthermore, due to the conciseness of the underlying algebra, definitions are simple and compact without having to deal with complicated indices in complex data structures.

In order to demonstrate the significance and usefulness of our framework, we have carried out experiments on existing distributed memory multiprocessors such as Intel's Paragon, and Touchstone Delta. Initially, our formal definitions are incorporated in three application problems: matrix transpose algorithm, two dimensional discrete Fourier transform algorithm, and solution of Euler partial differential equation using wavelet-Galerkin approach. Then, simple algebraic manipulations on these expressions are carried out to derive optimal data-partition and migration schemes. Experimental timing results on these machines show that such simple algebraic manipulations result in performance improvement ranging from 30% to 600%.

2.7 Existing Multidimensional FFT Algorithms

The Fourier transform of large multidimensional data sets is an essential computation in many scientific and engineering fields, including seismology, meteorology, x-ray crystallography, radar, sonar and medical imaging. Such fields require multidimensional arrays and large data set for complete and faithful modeling. The development of powerful parallel computers has given scientists a means of studying problems with greater complexity and higher dimensionality. Classically, a set of data is processed one dimension at a time, permitting control over the size of the computation and calling on well-established one-dimensional programs. Multidimensional processing offers a wider range of possible implementations as compared to one-dimensional processing, due to the greater flexibility of movement in the data indexing set. This increased freedom, along with large sized data sets typically found in multidimensional applications, places intensive demands on the communication aspects of the computation. Therefore, parallel programmers are facing greater challenges to develop efficient parallel FFT algorithms with minimum communication

overheads.

Because of its inherent algorithm structure, FFT lends itself naturally to parallel computation. There is a substantial amount of literature in parallelizing FFT, [28, 29, 30, 31, 32, 33, 34] to mention a few. In [28, 29, 30, 31] implementations of one-dimensional parallel FFTs on various multiprocessors were studied. In [32], implementation of high radix FFT on Boolean cube networks such as the Connection Machine was considered. Swarztrauber [34] investigated parallel FFT on general hypercube multiprocessors. He has derived an unordered parallel FFT algorithm on hypercube multiprocessors that has a minimum number of parallel data transfers between neighboring processors. This is performed by computing a one-dimensional FFT, which spans over processors. On the other hand, an existing two-dimensional FFT [35] in which FFTs on each dimension are computed by collecting all the required data and hence a computation is always within a node.

In Chapter 5, we presented an approach for computing multidimensional DFT on distributed memory systems that effectively utilizes the fact that today's distributed memory systems use wormhole routing for interprocessor communications. An approach to extend the algorithm for three or more dimensional problems using stride permutation and tensor product matrices has been presented that facilitates finding an efficient data-partitioning and network setup on distributed memory multiprocessors. Data-partitioning scheme is suitable and should be aimed at boundary value problems in fluid dynamics, finite element analysis etcetera. Results showed that our algorithm is more than six times as fast as the existing algorithm for certain cases. Moreover, higher the parallelism is, the better the performance of new algorithm will be. Given the fact that physical limits on memory exist at each processor, our new algorithm is a solution to today's large problems that involve multidimensional Fourier transform computations on massively parallel machines.

2.8 Survey of Matrix Algorithms

Many applications have numerical solutions in which computational burden is reduced partly or fully to matrix operations. One of the most elementary operations involving matrices is multiplication of two matrices. However, since matrix multiplication requires substantially more data movements than most other operations, algorithms that address efficient data movements are crucial to the effective implementation on concurrent computers.

For shared-memory systems, efficient parallel matrix computations are discussed in [36] with a theoretical package for parallel random access machines (PRAM). Without any optimization techniques, scalar operations in multiplication of two $N \times N$ matrices is in order of $O(N^3)$. However, Strassen [37] discovered an algorithm that only uses $O(N^{\log_2 7})$ scalar operations. Tensor product formulations of Strassen's algorithm are presented in [38, 39] along with capability to translate their mathematically equivalent tensor product formulations onto shared-memory architectures. Among algorithms constructed with an underlying topology in mind, Gentleman [40] has shown that for a mesh topology at least $0.35N$ routing steps are needed to compute product of two $N \times N$ matrices while $O(\log N)$ routes are necessary for a hypercube topology. For vector processors, Hayes [41] presented a technique at an element-by-element level for matrix-vector multiplication to solve systems of equations using iterative methods. She carried out implementation results on CYBER 205 demonstrating effectiveness for irregular problems.

For distributed memory systems, communication between nodes is done through message passings. If two or more nodes try to access same data, the requests will be queued and each node will get the message in a different time slot. This fact is pronounced in [42]. However, it is assumed in [43, 44] that simultaneous memory requests by all processors can be served within the same time slot. In Chapter 6, we reviewed Fox *et al'* s broadcast-and-shift matrix multiplication algorithm [25, 45] for message-passing architectures. This algorithm assumes mesh-division

data-partitioning for the underlying data. Multiplication algorithms in [42, 25, 45] require data movement of multiplicands irrespective of the size of the resulting matrix. Recently, Johnsson proposed an algorithm [46] to minimize the communication time in matrix multiplication. This algorithm is an evolution of considering two extreme cases of multiprocessor algorithm presented in [25, 45] but still requires data movement of multiplicands while results are accumulated in place.

Most existing research that has been reported in the literature for parallel matrix multiplications concentrates on mapping of the algorithm onto different topological structures. With the development of wormhole routing, algorithm performance becomes more sensitive to amount of data movements than the topological structures of the parallel machines being considered. Furthermore, all existing parallel matrix algorithms requires moving one or both multiplicands. In Chapter 6, we present an efficient algorithm [47] that requires no access of multiplicands by other processors due to the consideration of the unique data decomposition strategy. Only partial results need to be moved among processors. When compared to the algorithms in [25, 45, 46], messages in this algorithm are comparably shorter for the case of rectangular arrays. Performance improvements up to 440% have been observed over the algorithm in [25, 45] in actual implementations on Intel's Paragon and iPSC/860.

2.9 Experimental Environment

Intel's concurrent supercomputers are the cost-effective solution for large-scale applications. Experiments in this dissertation are performed on iPSC/860, Touchstone Delta, and Paragon supercomputers. All these systems consist of a set of processing nodes, I/O nodes, peripheral units, and a front-end processor. Each processing node uses one or more of the i860 multiprocessor. Message passing among nodes does not require any "store and forward" because of the Direct-Connect Module^{TM} (DCM). With the DCM, one can view these systems as an ensemble of fully connected nodes with a uniform message latency. This means that programmers do not have to structure their application's communication according to the underlying

topology (physical connections between nodes). On each node, a node system software runs to provide message-passing capabilities, memory management, and process management.

The iPSC/860 uses hypercube topology for physical connection to link 64 nodes. Each node is a processor/memory pair with memory size 8M bytes. The runtime software on iPSC is NX operating system. The Touchstone Delta uses mesh topology for physical connection to link 512 nodes. Again, each node is a processor/memory pair with possible memory sizes 8M bytes and 16M bytes. The runtime software on Touchstone Delta is NX/1 operating system. The Paragon also uses mesh connectivity to connect 64 nodes. However, each node has two i860 processors one to perform computations and another to perform the necessary communication instructions. Hence, Paragon provides higher communication bandwidth than iPSC or Delta. Memory assignment for each can be 16M bytes or 32M bytes. Also, Paragon uses a more advanced runtime software called OSF/1 provided by Open Software Foundation.

The runtime environment consists of a set of user interface commands that can be issued at the UNIX prompt and a set of system calls that are available to host and node programs. The most common programming model used with these supercomputers is the "single program, multiple data" (SPMD) model. In this model, the same program runs on each node in the application, but each node works on only part of the data. Any requirements of the data for a node from another node is obtained using their corresponding runtime software. Due to the underlying assembly coded routing-scheme and uniform message latency characteristics, we assumed in our analysis that a message-passing between two nodes, irrespective of the nodes, will have same communication time.

2.10 Conclusion

Notation involved in a mathematical language to express, to segregate an application, and to allocate tasks on parallel systems is explained with relevant theorems. Survey of existing data-partition schemes, existing multiprocessor Fourier transform and matrix multiplication algorithms is presented. A brief description about the platforms on which experiments in this dissertation are conducted is presented.

Chapter 3

Data Partition and Migration: Formal Definitions

3.1 Storing Data in Distributed Memories

Most large scale applications of scientific computing involve manipulations of data that are expressed in terms of matrices and vectors. This is natural because matrix notation gives a compact way to express computation. Moreover, storing matrices or vectors in the memory of a computer system is the first step of any computation. Different ways of storing data may result in different algorithmic structures as well as different computational performance. While methodology and algebraic formulations for storing matrices in a linear memory space of a single processor system exist, such as row-major and column-major, there is neither a formal and commonly agreed way of addressing data stored in distributed memory multiprocessor systems, nor an agreed formal description for various storage schemes. Programmers for parallel machines usually organize data in a way based on their convenience and efficiency of a specific algorithm. As a result, data-allocation and -partition in parallel processing are very diversified. Therefore, there is a need for a unified approach for formalizing data-allocation and -partitioning in parallel machines, and for a clear and convenient mathematical representation of various data-storage schemes. In parallel computers, particularly in distributed memory multiprocessors, communication costs are directly related to various data-storage schemes. Clear representation of storage schemes

helps parallel programmer greatly to look into structures of implementations and communication costs associated with algorithms.

Consider a message-passing multiprocessor system with k processors labeled from 0 to $k-1$, where $k = k_1\ k_2$. We would like to partition and store a two-dimensional (2D) matrix, denoted by $\mathbf{A}$ onto this system. For the purpose of simplicity and clarity of our presentation, we present only the cases [27] where the data can be evenly divided into k subsets and concentrate on our main interest of algebraic representation of partitioning the matrix and storing them into processors' memories.

Definition 3.1 Row-Division: *Let* $\mathbf{A}$ *be an* $M \times N$ *matrix. We define row-division onto* k *processors as follows. Partition* $\mathbf{A}$ *into* k *sets of complete rows such that* i*-th set of rows (top-down) is allocated to* i*-th processor. In matrix notation, row-division can be represented as operating by*

$$\mathbf{P}_R(M, N, k) = P(Nk, k) \otimes \mathbf{I}_{M/k} \tag{3.23}$$

on a vector $\mathbf{a}$ *that is formed as* $Vect_{MN}(\mathbf{A})$.

We use bold faced "P" ($\mathbf{P}$) with appropriate subscript to represent our data-partition definitions while italic "P" (P) to represent operation of stride permutation that explained in Section 2.3.

Definition 3.2 Column-Division: *Let* $\mathbf{A}$ *be an* $M \times N$ *matrix. We define column-division onto* k *processors as follows. Partitioning matrix* $\mathbf{A}$ *into* k *sets of complete columns such that* i*-th set of columns (left-right) is allocated to* i*-th processor. In matrix notation, column-division is represented as operating by*

$$\mathbf{P}_C(M, N, k) = \mathbf{I}_{MN} \tag{3.24}$$

on a vector $\mathbf{a}$ *that is formed as* $Vect_{MN}(\mathbf{A})$.

Definition 3.3 Mesh-Division: *Let* $\mathbf{A}$ *be an* $M \times N$ *matrix. We define mesh-division of* $\mathbf{A}$ *onto a system with* $k_1 \times k_2$ *processors as follows. Partition* M *rows*

of $\mathbf{A}$ *into* k_1 *equal sets of rows (top-down) and then partition each set of rows into* k_2 *equal subsets (left-right). Each subset is a* $M/k_1 \times N/k_2$ *size matrix but will have neither complete rows nor complete columns. Allocation of these subsets to* k *processors is performed anti-lexicographically (top-down and then left-right). In matrix notation, mesh-division is defined as*

$$\mathbf{P}_M(M, N, k_1, k_2) = \mathbf{I}_{k_2} \otimes P(Nk_1/k_2, k_1) \otimes \mathbf{I}_{M/k_1}. \tag{3.25}$$

Following three equations represent inverse operations of the above three definitions which can be derived using theorems 2.5 and 2.9.

$$\mathbf{P}_R^{-1}(M, N, k) = P(Nk, N) \otimes \mathbf{I}_{M/k} \tag{3.26}$$

$$\mathbf{P}_C^{-1}(M, N, k) = \mathbf{I}_{MN} \tag{3.27}$$

$$\mathbf{P}_M^{-1}(M, N, k_1, k_2) = \mathbf{I}_{k_2} \otimes P(Nk_1/k_2, N/k_2) \otimes \mathbf{I}_{M/k_1} \tag{3.28}$$

Example 3.1 *This example demonstrates data partitioning of an* 8×8 *matrix,* $\mathbf{A}$*, onto a 4-processor machine. Figure 3.1 shows how a 64-element vector* $\mathbf{a}$ *formed by* $Vect_{64}(\mathbf{A})$ *is partitioned in row-division, column-division, and mesh-division based on Definitions 3.1-3.3. In case of row-division,* $\mathbf{I}_2$ *on the right-hand side of* $\mathbf{P}_R(8, 8, 4)$ *represents moving vectors of length 2 according to the permutation matrix* $P(32, 4)$*. When this permutation is applied, resulting data at processor-0 is shown with dotted-line. For column-division data partitioning, since input permutation is an identity matrix, no action needs to be performed, and the vector* $\mathbf{a}$ *is just segmented into four parts for allocating to four processors. For mesh-division data partitioning,* $\mathbf{I}_2$ *on the left-hand side of* $\mathbf{P}_M(8, 8, 2, 2)$ *represents an action to divide the vector* $\mathbf{a}$ *into two equal sets and perform the vector-stride action* $P(8, 2) \otimes \mathbf{I}_4$ *on each set. However, this vector-stride further divides each set into eight small subvectors of length 4 and shuffle them according to the permutation* $P(8, 2)$*. Once again, data residing at processor-0 after the action of input permutation is shown with dotted-line.*

General Usage of Data-Partition Definitions

Consider any computational procedure that is expressed by an operational matrix

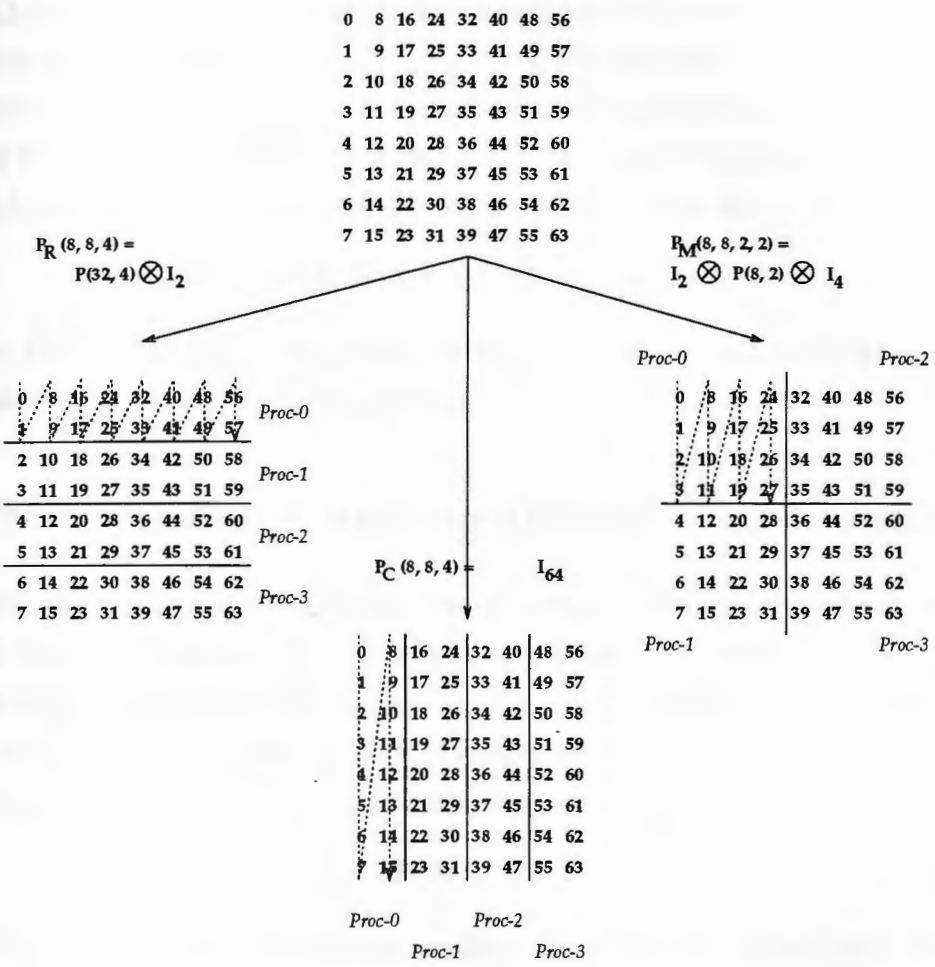


Figure 3.1: Action of data-partition algebraic expressions onto a 4-processor machine

$\mathbf{G}$ operating on a vector $\mathbf{a}$ to obtain vector $\mathbf{b}$:

$$\mathbf{b} = \mathbf{G}\,\mathbf{a}. \tag{3.29}$$

This equation ignores the underlying data-partition necessary to carryout the computation in distributed memory multiprocessor system. To bring out the data-partition, let $\hat{\mathbf{a}}\,(= \mathbf{Q}_1\mathbf{a})$ be a desired data partition of $\mathbf{a}$ among the processors where $\mathbf{Q}_1$ is one of the data partition schemes ($\mathbf{P}_R$, $\mathbf{P}_C$, or $\mathbf{P}_M$) defined above. If one expects the output data to be in a particular partition after the computation, then resultant data is of the form $\hat{\mathbf{b}}$ where $\hat{\mathbf{b}} = \mathbf{Q}_2\mathbf{b}$ and $\mathbf{Q}_2$ is also one of the definitions $\mathbf{P}_R$, $\mathbf{P}_C$, or $\mathbf{P}_M$ defined above. Then equation (3.29) can be rewritten as:

$$\hat{\mathbf{b}} = \mathbf{Q}_2\,\mathbf{b} = \mathbf{Q}_2\,\mathbf{G}\,\mathbf{a} = \left[\mathbf{Q}_2\,\mathbf{G}\,\mathbf{Q}_1^{-1}\right]\hat{\mathbf{a}} \stackrel{\text{def}}{=} \hat{\mathbf{G}}\,\hat{\mathbf{a}} \tag{3.30}$$

Therefore, $\hat{\mathbf{G}} = \mathbf{Q}_2\,\mathbf{G}\,\mathbf{Q}_1^{-1}$ is the actual-operational matrix that takes into account the complexity of considered data partition.

3.2 Moving Data Among Distributed Memories

Once input data is partitioned among the processors, data migrations at the interfaces between individual algorithms may be necessary in order to achieve global optimal performance of an application. One frequently used data migration in numerical applications is well known matrix transpose. Let $\mathbf{a} = Vect_{MN}(\mathbf{A}_{M\times N})$, and $\mathbf{b} = Vect_{NM}(\mathbf{B}_{N\times M})$, where $\mathbf{B}_{N\times M}$ is the transpose of $\mathbf{A}_{M\times N}$. Then,

$$\mathbf{b} = P(MN, M)\,\mathbf{a}. \tag{3.31}$$

Hence $P(MN, M)$ is the operational matrix for transpose algorithms, that is, $\mathbf{G} = P(MN, M)$. When data partition schemes are to be incorporated, the actual-operational matrix becomes $\hat{\mathbf{G}}$ (see equation (3.30)). That is,

$$P(MN, M) = \mathbf{Q}_2^{-1}\,\hat{\mathbf{G}}\,\mathbf{Q}_1, \tag{3.32}$$

and the equation (3.31) becomes

$$\hat{\mathbf{b}} = \hat{\mathbf{G}}\,\hat{\mathbf{a}}, \tag{3.33}$$

where $\hat{\mathbf{G}} = \mathbf{Q}_2 P(MN, M)\mathbf{Q}_1^{-1}$. In the following, we will show how to derive the operational matrices, $\hat{\mathbf{G}}$, required to transpose a matrix using the data-partitions defined in previous section (assume $\mathbf{Q}_1 = \mathbf{Q}_2$ for simplicity) and discuss their implementation aspects via the tensor product formulations.

ROW-DIVISION

For row-division data partition, we have

$$\hat{\mathbf{G}} = \mathbf{P}_R(N, M, k)P(MN, M)\mathbf{P}_R^{-1}(M, N, k). \tag{3.34}$$

According to Definition 3.1, we have

$$\hat{\mathbf{G}} = \left[P(Mk, k) \otimes \mathbf{I}_{N/k}\right] P(MN, M) \left[P(Nk, N) \otimes \mathbf{I}_{M/k}\right], \tag{3.35}$$

(or)

$$\mathbf{G} = P(MN, M) = \left[P(Mk, M) \otimes \mathbf{I}_{N/k}\right] \hat{\mathbf{G}} \left[P(Nk, k) \otimes \mathbf{I}_{M/k}\right]. \tag{3.36}$$

Then, we can obtain expression for $\hat{\mathbf{G}}$ by dissecting $\mathbf{G} = P(MN, M)$ as:

$$P(MN, M) = \left[P(Mk, M) \otimes \mathbf{I}_{N/k}\right] \left[\mathbf{I}_k \otimes P(MN/k, M)\right]$$

by theorem 2.7

$$P(MN, M) = \mathbf{P}_R^{-1}(N, M, k) \left[\mathbf{I}_{k^2} \otimes P(MN/k^2, M/k)\right] \left[(\mathbf{I}_k \otimes P(N, k)) \otimes \mathbf{I}_{M/k}\right]$$

by theorem 2.8 and equation (3.26)

$$P(MN, M) = \mathbf{P}_R^{-1}(N, M, k) \left[\mathbf{I}_k \otimes \mathbf{I}_k \otimes P(MN/k^2, M/k)\right] \left[P(k^2, k) \otimes \mathbf{I}_{N/k} \otimes \mathbf{I}_{M/k}\right] \left[P(Nk, k) \otimes \mathbf{I}_{M/k}\right]$$

by applying theorem 2.7 to$P(Nk, k)$

$$P(MN, M) = \mathbf{P}_R^{-1}(N, M, k) \left[\mathbf{I}_k \otimes \mathbf{I}_k \otimes P(MN/k^2, M/k)\right] \left[P(k^2, k) \otimes \mathbf{I}_{MN/k^2}\right] \mathbf{P}_{in}(\text{row}, M, N, k) \tag{3.37}$$

by Definition 3.1

$$P(MN, M) = \mathbf{P}_R^{-1}\ \hat{\mathbf{G}}\ \mathbf{P}_R.$$

```
me = my node number
for index = 1 to k − 1
   myswap = xor(me,index)
   Send block-myswap of my associated vector a to processor-myswap
   Receive message from processor-myswap
   Store message at block-myswap of my associated vector a
end
```

Table 3.2: Pseudo-code for message passing in transpose algorithms for either row-division or column-division partitions

Therefore, the actual-operational matrix in equation (3.33) for row-division partition can be expressed as two stages:

$$\hat{\mathbf{G}} = \left[\mathbf{I}_k \otimes \mathbf{I}_k \otimes P(MN/k^2, M/k)\right] \left[P(k^2, k) \otimes \mathbf{I}_{MN/k^2}\right]. \tag{3.38}$$

The first stage, $\underline{P(k^2, k) \otimes \mathbf{I}_{MN/k^2}}$, is a *global-task* that involves message-passings among processors since the expression does not contain an identity matrix, $\mathbf{I}_k$, on left-hand side. The size of each message being passed is (MN/k^2) which is $(1/k)$th of the size of the data set residing at a processor. This is reflected in the above tensor product expression by $\mathbf{I}_{MN/k^2}$. The factor $P(k^2, k)$ in the expression indicates that each processor has $(k-1)$ subblocks to send out. Such message passings are carried out in $(k-1)$ stages with one subblock being kept within a processor. The pseudo-code implementation for this stage is shown in Table 3.2.

The second stage, $\underline{\mathbf{I}_k \otimes \mathbf{I}_k \otimes P(MN/k^2, M/k)}$, represents a *local-task* due to the identity matrix $\mathbf{I}_k$ on the left-hand side. Each processor performs the parallel-stride operation $[\mathbf{I}_k \otimes P(MN/k^2, M/k)]$ locally.

COLUMN-DIVISION

For column-division data partition, we have

$$\hat{\mathbf{G}} = \mathbf{P}_C(N, M, k) P(MN, M) \mathbf{P}_C^{-1}(M, N, k) \tag{3.39}$$

According to Definition 3.2, we have

$$\hat{\mathbf{G}} = \mathbf{I}_{MN} P(MN, M) \mathbf{I}_{MN} = P(MN, M) = \mathbf{G}. \tag{3.40}$$

Then, we can obtain expression for $\hat{\mathbf{G}}$ as:

$$P(MN, M) = [\mathbf{I}_k \otimes P(MN/k, M/k)] \left[P(Nk, k) \otimes \mathbf{I}_{M/k}\right]$$

by theorem 2.7

$$P(MN, M) = [\mathbf{I}_k \otimes P(MN/k, M/k)] \left[\left\{\left(P(k^2, k) \otimes \mathbf{I}_{N/k}\right)(\mathbf{I}_k \otimes P(N, k))\right\} \otimes \mathbf{I}_{M/k}\right]$$

by theorem 2.8

$$P(MN, M) = [\mathbf{I}_k \otimes P(MN/k, M/k)] \left[P(k^2, k) \otimes \mathbf{I}_{MN/k^2}\right] \left[\mathbf{I}_k \otimes P(N, k) \otimes \mathbf{I}_{M/k}\right]. \tag{3.41}$$

Therefore, the actual-operational matrix in equation (3.33) for column-division partitioning can be expressed as three stages:

$$\hat{\mathbf{G}} = [\mathbf{I}_k \otimes P(MN/k, M/k)] \left[P(k^2, k) \otimes \mathbf{I}_{MN/k^2}\right] \left[\mathbf{I}_k \otimes P(N, k) \otimes \mathbf{I}_{M/k}\right] \tag{3.42}$$

The first stage, $\underline{\mathbf{I}_k \otimes P(N, k) \otimes \mathbf{I}_{M/k}}$, represents *local* data permutations without message-passing due to the prior identity $\mathbf{I}_k$. Each processor performs the vector-stride operation $\left[P(N, k) \otimes \mathbf{I}_{M/k}\right]$ which moves N vectors with stride k, each vector is of length (M/k).

The second stage, $\underline{P(k^2, k) \otimes \mathbf{I}_{MN/k^2}}$, is a *global-task* that is similar to message-passing stage explained in row-division transpose algorithm. Hence the total communication is again $(k-1)$ messages, each message is of length (MN/k^2).

The final stage, $\underline{\mathbf{I}_k \otimes P(MN/k, M/k)}$, is a and simple-stride permutation stage with stride (M/k) *local* to each processor. All processors carry out the same operation in parallel without communication.

MESH-DIVISION

For mesh-division partition, we have

$$\hat{\mathbf{G}} = \mathbf{P}_M(N, M, k_2, k_1) P(MN, M) \mathbf{P}_M^{-1}(M, N, k_1, k_2). \tag{3.43}$$

According to Definition 3.3, we have

$$\hat{\mathbf{G}} = \left[\mathbf{I}_{k_1} \otimes P(Mk_2/k_1, k_2) \otimes \mathbf{I}_{N/k_2}\right] P(MN, M) \left[\mathbf{I}_{k_2} \otimes P(Nk_1/k_2, N/k_2) \otimes \mathbf{I}_{M/k_1}\right], \tag{3.44}$$

(or)

$$P(MN, M) = \left[\mathbf{I}_{k_1} \otimes P(Mk_2/k_1, M/k_1) \otimes \mathbf{I}_{N/k_2}\right] \hat{\mathbf{G}} \left[\mathbf{I}_{k_2} \otimes P(Nk_1/k_2, k_1) \otimes \mathbf{I}_{M/k_1}\right]. \tag{3.45}$$

Then we can obtain expression for $\hat{\mathbf{G}}$ by decomposing $\mathbf{G} = P(MN, M)$ as follows.

$$P(MN, M) = [\mathbf{I}_{k_1} \otimes P(MN/k_1, M/k_1)] \left[P(Nk_1, k_1) \otimes \mathbf{I}_{M/k_1}\right]$$

by theorem 2.7

$$P(MN, M) = \left[\mathbf{I}_{k_1} \otimes P(Mk_2/k_1, M/k_1) \otimes \mathbf{I}_{N/k_2}\right] [\mathbf{I}_k \otimes P(MN/k, M/k_1)] \left[P(k, k_1) \otimes \mathbf{I}_{MN/k}\right] \left[\mathbf{I}_{k_2} \otimes P(Nk_1/k_2, k_1) \otimes \mathbf{I}_{M/k_1}\right] \tag{3.46}$$

by theorem 2.8 on $P(MN/k_1, M/k_1)$and by theorem 2.7 on $P(Nk_1, k_1)$

$$P(MN, M) = \mathbf{P}_M^{-1}(N, M, k_2, k_1) [\mathbf{I}_k \otimes P(MN/k, M/k_1)] \left[P(k, k_1) \otimes \mathbf{I}_{MN/k}\right] \mathbf{P}_M(M, N, k_1, k_2) \tag{3.47}$$

by equation (3.28) and Definition 3.3

$$P(MN, M) = \mathbf{P}_M^{-1}(N, M, k_2, k_1) \left[P(k, k_1) \otimes \mathbf{I}_{MN/k}\right] [\mathbf{I}_k \otimes P(MN/k, M/k_1)] \mathbf{P}_M(M, N, k_1, k_2) \tag{3.48}$$

by commutative law

$$P(MN, M) = \mathbf{P}_M^{-1} \hat{\mathbf{G}} \mathbf{P}_M$$

Therefore, the actual-operational matrix in equation (3.33) for mesh-division partition can be expressed as two stages in two different ways (equations (3.47) and (3.48)).

(a) $\hat{\mathbf{G}} = [\mathbf{I}_k \otimes P(MN/k), M/k_1)] \left[P(k, k_1) \otimes \mathbf{I}_{MN/k}\right]$, and

(b) $\hat{\mathbf{G}} = \left[P(k, k_1) \otimes \mathbf{I}_{MN/k}\right] [\mathbf{I}_k \otimes P(MN/k, M/k_1)]$.

In case of (a), the first stage, $\underline{P(k, k_1) \otimes \mathbf{I}_{MN/k}}$, is a *global-task* involving message-passings since there is no prior identity matrix. In fact, it is a single message-passing

M	N	Row-Division (msec)	Col-Division (msec)	Mesh-Division (msec)
128	128	5.236	6.172	1.316
128	256	5.902	7.051	2.028
128	512	9.031	10.409	2.159
128	1024	12.356	15.312	3.866
256	128	5.501	6.665	1.825
256	256	8.283	9.746	2.301
256	512	11.483	14.027	4.018
256	1024	20.076	22.503	7.548
512	128	8.310	9.432	3.450
512	256	11.555	13.359	5.905
512	512	18.536	21.122	7.954
512	1024	39.628	38.529	16.434
1024	128	11.228	13.132	5.815
1024	256	17.526	20.616	10.631
1024	512	31.211	37.445	20.889
1024	1024	50.936	66.403	49.274

Table 3.3: Experimental results of transpose algorithms on Intel's Paragon

routine with message size being (MN/k) as compared to $(k-1)$ messages each of size (MN/k^2) in either row-division or column-division transpose algorithms.

The second stage, $\underline{\mathbf{I}_k \otimes P(MN/k, M/k_1)}$, represents that each processor executes a *local* simple-stride permutation because of prior identity matrix $\mathbf{I}_k$. In fact, if we consider data at each processor to be a matrix of size $M/k_1 \times N/k_2$, then action to be performed in this stage is k local matrix transposes that are performed simultaneously on k processors.

3.2.1 Performance Evaluation of Three Transpose Algorithms

Transpose algorithms derived in Section 3.2 are implemented on Intel's Paragon and Touchstone Delta, and results are tabulated in Tables 3.3 and 3.4, respectively.

M	N	Row-Division (msec)	Col-Division (msec)	Mesh-Division (msec)
128	128	8.092	8.865	2.681
128	256	10.042	12.280	5.769
128	512	13.988	18.980	11.702
128	1024	23.909	33.014	20.018
256	128	10.065	12.016	5.041
256	256	14.228	18.150	11.554
256	512	23.030	31.237	20.088
256	1024	43.458	59.109	36.009
512	128	13.982	17.920	9.822
512	256	23.002	30.593	19.637
512	512	44.178	57.799	36.091
512	1024	95.145	114.215	79.681
1024	128	22.743	30.400	19.507
1024	256	42.197	57.171	36.109
1024	512	83.011	113.416	79.492
1024	1024	187.484	223.287	167.497

Table 3.4: Experimental results of transpose algorithms on Intel's Touchstone Delta

From the derivations in equations (3.38), (3.41), (3.47), and (3.48), we have seen that to transpose a matrix of size $M \times N$ on a k-processor machine for row-division and column-division partitionings each requires $(k-1)$ number of communications, each communication is of size (MN/k^2) while mesh-division partitioning requires *one* communication that is of size (MN/k). Though message length in mesh-division is k times more than that of any message in either row-division or column-division, results in Tables 3.3 and 3.4 clearly show that transpose algorithm for mesh-division eliminates the overheads to initiate a communication. These results also show that unlike uniprocessor algorithms, variations in data-decompositions can have a great impact on the performance of an algorithm.

3.3 An Example

As an example of applying our definition of data partitioning and migration, this section presents a typical two-dimensional (2D) fast Fourier transform (FFT) algorithm in a distributed memory system using our new formalism [48].
The summation form of the 2D-DFT on a matrix $\mathbf{X}$ of size $M \times N$ is given by:

$$\mathbf{Y}(k,l) = \sum_{m=0}^{M-1} \left[\sum_{n=0}^{N-1} \mathbf{X}(m,n) e^{-j\frac{2\pi nl}{N}} \right] e^{-j\frac{2\pi mk}{M}} \tag{3.49}$$

while tensor product representation of equation (3.49) can be written as:

$$\mathbf{y} = \underbrace{[\mathbf{F}_N \otimes \mathbf{F}_M]}_{\mathbf{G}} \mathbf{x}, \tag{3.50}$$

where $\mathbf{F}_J$ is a $J \times J$ matrix with ith row, jth column entry equals to $\exp(-j2\pi ik/J)$, $j = \sqrt{-1}$, $\mathbf{y} = Vect_{MN}(\mathbf{Y})$, $\mathbf{x} = Vect_{MN}(\mathbf{X})$, and $\mathbf{G}$ is the operational matrix.

To compute the equation (3.50) on a k-processor parallel machine, we first parallelize the operational matrix by inserting identity matrices, assuming k divides both M and N. There are two ways of decomposing the equation (3.50): (a) $\mathbf{y} = [\mathbf{I}_N \otimes \mathbf{F}_M][\mathbf{F}_N \otimes \mathbf{I}_M]\ \mathbf{x}$ which first computes Fourier transforms of columns followed by transforms of rows, and (b) $\mathbf{y} = [\mathbf{F}_N \otimes \mathbf{I}_M][\mathbf{I}_N \otimes \mathbf{F}_M]\ \mathbf{x}$, which performs transformation on rows followed by that on columns. Consider the first decomposition (a). The factor on the left-hand side represents a parallel computation of $\mathbf{F}_M$ because of preceding identity matrix $\mathbf{I}_N$ while the factor on the right-hand side cannot be done in parallel. To parallelize this stage of computation, we apply the commutative law presented in theorem 2.4, resulting in

$$\mathbf{y} = [\mathbf{I}_N \otimes \mathbf{F}_M]\ P(MN,N)\ [\mathbf{I}_M \otimes \mathbf{F}_N]\ P(MN,M)\ \mathbf{x} \tag{3.51}$$

If it is required that the Fourier transformed data be in the same data-partition scheme as the original data, then input data is $\hat{\mathbf{x}} = \mathbf{P}_R\ \mathbf{x}$ and output data is $\hat{\mathbf{y}} = \mathbf{P}_R\ \mathbf{y}$. In such a case, equations (3.51) can be rewritten as:

$$\hat{\mathbf{y}} = \mathbf{P}_R\ [\mathbf{I}_N \otimes \mathbf{F}_M]\ P(MN,N)\ [\mathbf{I}_M \otimes \mathbf{F}_N]\ [P(MN,M)\ \mathbf{P}_R^{-1}]\ \hat{\mathbf{x}}. \tag{3.52}$$

Note that if we used second parallelization (b), we would have obtained

$$\hat{\mathbf{y}} = [\mathbf{P}_R P(MN, N)] [\mathbf{I}_M \otimes \mathbf{F}_N] P(MN, M) [\mathbf{I}_N \otimes \mathbf{F}_M] \mathbf{P}_R^{-1} \hat{\mathbf{x}}. \tag{3.53}$$

In the following, we will see how we utilize our new definitions on data partition and migration to maximize the parallelism and minimize the communication cost while computing equation (3.52).

Consider row-division partitioning and start with the first stage operator (the right most factor) of equation (3.52). Recall that $\mathbf{P}_R^{-1} = \left[P(Nk, N) \otimes \mathbf{I}_{M/k}\right]$ from Equation (3.23). It appears from equation (3.52) that neither $\mathbf{P}_R^{-1} = \left[P(Nk, N) \otimes \mathbf{I}_{M/k}\right]$ nor $P(MN, M)$ has a prior identity matrix $\mathbf{I}_k$ implying that both operations involve message-passings. However, simple algebra manipulations of this computation stage based on our definitions can lead to a completely parallel computation. Decomposing $P(MN, M)\mathbf{P}_R^{-1}$ in a different way, we have

$$\begin{aligned} P(MN, M)\mathbf{P}_R^{-1} &= P(k(M/k)N, k(M/k))\mathbf{P}_R^{-1} \\ &= [\mathbf{I}_k \otimes P(MN/k, M/k)] \left[P(Nk, k) \otimes \mathbf{I}_{M/k}\right] \mathbf{P}_R^{-1} \\ &= \underbrace{[\mathbf{I}_k \otimes P(MN/k, M/k)]}_{\mathcal{Z}_1} \end{aligned} \tag{3.54}$$

No communication! For notational convenience, we use $\mathcal{Z}_i$ to denote the ith computation stage. We will see shortly in this section that all the involved computation stages are directly computable using the existing subroutines in machine libraries of a commercial distributed memory multiprocessor.

The second stage of computation in equation (3.52) is obviously a parallel computation because of $\mathbf{I}_M$. The next stage, $P(MN, N)$, can be factored as

$$\begin{aligned} P(MN, N) &= \left[P(Nk, N) \otimes \mathbf{I}_{M/k}\right] \underbrace{[\mathbf{I}_k \otimes P(MN/k, N)]}_{\mathcal{Z}_3} \\ &= \left[\left\{[\mathbf{I}_k \otimes P(N, N/k)] \left[P(k^2, k) \otimes \mathbf{I}_{N/k}\right]\right\} \otimes \mathbf{I}_{M/k}\right] \mathcal{Z}_3 \end{aligned}$$

$$= \left[\mathbf{I}_k \otimes P(N, N/k) \otimes \mathbf{I}_{M/k}\right] \underbrace{\left[P(k^2, k) \otimes \mathbf{I}_{MN/k^2}\right]}_{\mathcal{Z}_4} \mathcal{Z}_3$$

$$= \underbrace{\left[\mathbf{I}_k \otimes P(MN/k, N/k)\right]}_{\mathcal{Z}_6} \underbrace{\left[\mathbf{I}_k \otimes \mathbf{I}_k \otimes P(MN/k^2, M/k)\right]}_{\mathcal{Z}_5} \mathcal{Z}_4\mathcal{Z}_3 \quad (3.55)$$

Note that $P(MN, N) = \left[P(Nk, N) \otimes \mathbf{I}_{M/k}\right] \left[\mathbf{I}_k \otimes P(MN/k, N)\right]$ according to theorem 2.8. From the definition of $\mathbf{P}_R^{-1}(M, N, k)$ we have

$$P(MN, N) = \mathbf{P}_R^{-1} \left[\mathbf{I}_k \otimes P(MN/k, N)\right].$$

Left multiplying $\mathbf{P}_R$ and right multiplying $P(MN, M)$ on both sides of above equation, we obtain

$$\mathbf{P}_R = \left[\mathbf{I}_k \otimes P(MN/k, N)\right] P(MN, M)$$

To expand $P(MN, M)$, we use equation (3.55) but interchange the roles of M and N. We have

$$\mathbf{P}_R = \underbrace{\left[\mathbf{I}_k \otimes P(MN/k, N)\right] \left[\mathbf{I}_k \otimes P(MN/k, M/k)\right]}_{\mathbf{I}_{MN}} \underbrace{\left[I_k \otimes \mathbf{I}_k \otimes P(MN/k^2, N/k)\right]}_{\mathcal{Z}_{10}}$$
$$\underbrace{\left[P(k^2, k) \otimes \mathbf{I}_{MN/k^2}\right]}_{\mathcal{Z}_9} \underbrace{\left[\mathbf{I}_k \otimes P(MN/k, M)\right]}_{\mathcal{Z}_8} \quad (3.56)$$

Now we summarize the above decompositions and recombine them in association with the three submodules (a) **bfft**, (b) **global23**, and (c) **local12** that are available in a library in Intel's supercomputers.

$$\hat{\mathbf{y}} = [\mathbf{P}_R] \underbrace{[\mathbf{I}_N \otimes \mathbf{F}_N]}_{\mathcal{Z}_7} [P(MN, N)] \underbrace{[\mathbf{I}_M \otimes \mathbf{F}_N]}_{\mathcal{Z}_2} \left[P(MN, M)\mathbf{P}_R^{-1}\right] \hat{\mathbf{x}}$$

$$= [\mathcal{Z}_{10}\mathcal{Z}_9\mathcal{Z}_8] [\mathcal{Z}_7] [\mathcal{Z}_6\mathcal{Z}_5\mathcal{Z}_4\mathcal{Z}_3] [\mathcal{Z}_2] [\mathcal{Z}_1] \hat{\mathbf{x}}$$

$$= \underbrace{[\mathcal{Z}_{10}]}_{\textsf{local12}} \underbrace{[\mathcal{Z}_9]}_{\textsf{global23}} \underbrace{[\mathcal{Z}_8\mathcal{Z}_7\mathcal{Z}_6]}_{\textsf{bfft}} \underbrace{[\mathcal{Z}_5]}_{\textsf{local12}} \underbrace{[\mathcal{Z}_4]}_{\textsf{global23}} \underbrace{[\mathcal{Z}_3\mathcal{Z}_2\mathcal{Z}_1]}_{\textsf{bfft}} \hat{\mathbf{x}} \quad (3.57)$$

Module Description:

local12: Local Permutations

bfft: Local permutations + (N/k) number of M-point FFTs or (M/k) number of N-point FFTs + local permutations.

global23: Block transpose algorithm involving $(k-1)$ number of node-to-node communications each of size (MN/k^2) as seen in transpose algorithms.

3.4 Comparison of Our Definitions with Related Work

Data organization is the key to successful parallelization of data parallel programs. As indicated in the introduction, there are two tracks of efforts in data-partition and migration in distributed memory multiprocessors: automatic data-partitioning for general loop constructs as part of compiler and optimal partitioning for a specific algorithm. In this section, we briefly summarize the existing works in this field as related to our work presented in this dissertation. For more comprehensive review of previous work in data-partitioning and redistribution, readers are referred to [17, 16, 19].

Ramanujam and Sadayappan [16] studied compile-time techniques for data-partitioning in distributed memory systems. They presented an analysis of communication-free partitions with a nice geometric demonstration. The research work performed by Li and Chen [20] focused on minimizing data movement among processors due to cross-references of multiple distributed arrays (alignment of multiple data structures). They have also presented a method of automatically generating efficient message-passing routines in parallel programs [20]. Gupta and Banerjee introduced the notion of constraints on data-partitioning to obtain good performance. In [23], a compiler algorithm was described to automatically finds optimal parallelism and optimal locality in general loop nesting. All these studies aimed at optimizing

data-partition and data alignments as part of compiler. It is known that such optimization problem is NP-complete. A number of heuristics have been proposed [20, 21, 22, 16, 49, 15].

The use of tensor product notation to describe parallel algorithms has a long history beginning with Pease [50]. Johnson *et al* [33] presented a comprehensive discussion on how to use tensor notations to design, modify and implement FFT algorithms on various computer architectures. Attempts to derive variants of FFT algorithms keeping the underlying architecture in mind have proven successful [24, 13]. Huang, Johnson and Johnson [38] have recently used tensor notations for formulating Strassen's matrix multiplication algorithm. Using the tensor representation, they derived three variant programs and compared their performance characteristics for shared memory multiprocessors.

Kaushik, Huang, Johnson and Sadayappan have proposed a very nice approach for data redistribution in distributed memory systems, which appeared recently in [19]. While their approach also utilizes the tensor notation as a tool, our work differs in several aspects. First of all, our definitions are expressed in matrix forms while theirs are in terms of indices (tensor bases). With their model one can estimate communication cost of a computation precisely while with our formulations one can easily manipulate the communication structures of a computation to achieve optimal performance. Deriving variants of an algorithm using our definitions are relatively simple because the data communication is easily visible. Secondly, all the definitions presented in [19] such as *cyclic, block*, and *block cyclic* can be defined using our formulations as evidenced in Section 3, whereas some of data-partitions such as mesh-division cannot be easily expressed using the notations in [19]. In addition, our representation acts directly on data vector $\mathbf{a}(O : N-1)$ to achieve the required data-partition and migration scheme while their representation presents ways to manipulate data indices from one distribution to the other (redistribution). Unlike their representation, we can embed our expressions for data distribution into an algorithm. As a result, global optimization of an application consisting of several computation

modules become straightforward by just manipulating the algebraic expression at the interfaces between individual algorithms.

3.5 Conclusion

This chapter introduced a unified approach for formalizing data-allocation and -partitioning in parallel machines, and for a clear and convenient mathematical expressions of various data storage schemes. These expressions are successfully used in expressing, deriving, and implementing matrix transpose algorithms. In turn, expressions derived for transpose algorithms are used in representation of existing multiprocessor two-dimensional FFT algorithm. Representation for existing three-dimensional FFT algorithm at Intel's supercomputers is presented in Appendix A.

Chapter 4

Switching Data Partition Schemes Within An Application

4.1 Introduction

In this chapter we consider the implementation of an application in which we solve Euler partial differential equation (PDE) for two-dimensional case using wavelet-Galerkin method [26, 51, 52]. The two most important computation modules in this solution require two different data-partitions for their optimal implementation. Such a situation demands switching data-partition that might involve message-passing stages among processors. This chapter evidences benefit of the algebraic expressions defined in Chapter 3 in such a situation.

First module, *Helmholtz*, involves two-dimensional filtering with forward and inverse two-dimensional Fourier transform (2D-FFT) techniques. It is seen in previous chapter that efficient multiprocessor 2D-FFT algorithm exists for row-division or column-division data-partitioning.

The second module computes *Jacobian* that consists of numerous small intra-node matrix multiplications. The module Jacobian requires boundary data from other nodes, but if we neglect for the moment the necessity for neighboring spatial regions to exchange data, choice of any data-partitioning shows ideal concurrency,

with no sequential dependence of one processor's calculation on other's. Departure from ideal speedup in evolution of Jacobian arises because the elements on four sides, considering mesh-division data-partition, or the elements on two sides if we consider either row or column-division data-partition, of any particular processor require boundary elements from its geometrically neighboring processors.

Therefore, Jacobian is optimal for mesh-division data-partition considering the fact that size of data to be shared by other processors is less compared to the size of data to be shared by other processors in either row- or column-division data-partitions. To make use of the peak performances of these modules individually, switching between row-division and mesh-division data-partitions would be an overhead. This chapter deals with the minimization of this overhead by combining the data-partition expressions with the 2D-FFT algorithms.

Section 4.2 presents a brief description of an algorithm to solve a PDE using wavelet-Galerkin method (see flowchart shown in Figure 4.2). Readers interested in details of wavelet-Galerkin method can refer to [26, 52]. Then, Sections 4.3.1 and 4.3.2 derive two variants of two-dimensional FFT algorithms that start from mesh-division data-partitioning but go through column-division partitioning to retain the efficiency of message-passing in row- or column-division partition, and come back to mesh-division data-partitioning to compute Jacobian. Section 4.4 presents implementation results of these FFT algorithms compared to the FFT algorithm seen in Section 3.3 for row-division partition. It also presents results on overall performance on application which show up to 43.61% reduction in time.

4.2 Brief Description of Application

Since we are not interested in exact computational details here but switching data-partition among the modules, we present only the required details and concentrate on data structure compatibility. Figure 4.2 shows the flowchart for evaluating the coefficients for vorticity in fluid mechanics at each time-step. In this procedure, major

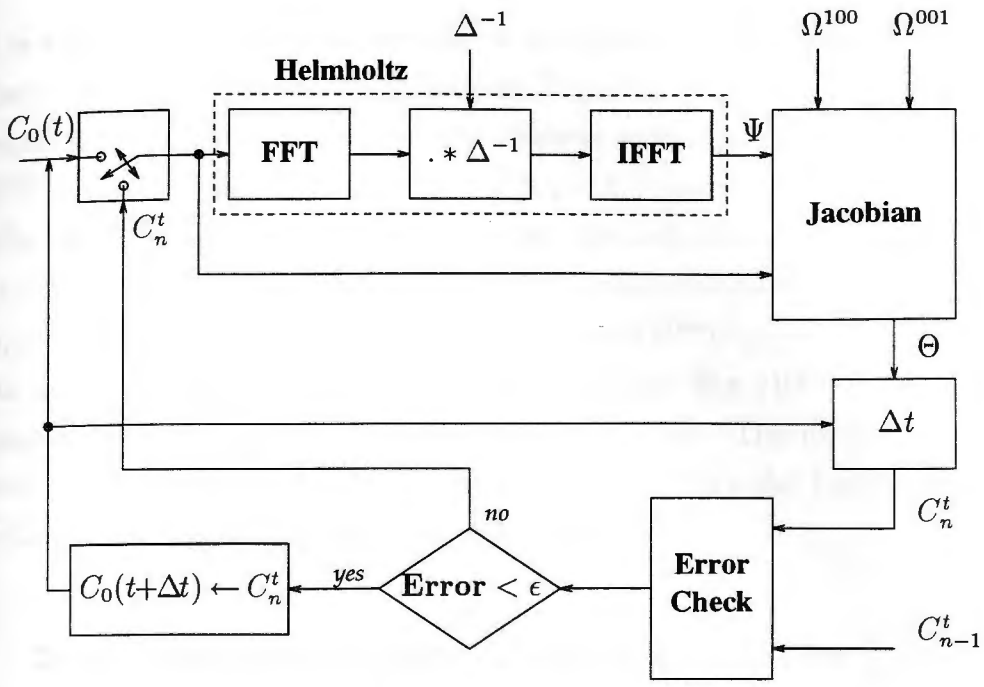


Figure 4.2: Flow Chart for computation of coefficients of Vorticity

computation blocks are Jacobian and Helmholtz, and minor computation modules are Error Check, and the computation of vorticity coefficients in next step (Δt).

The module Jacobian results in the so called wavelet-Galerkin operator Θ that depends on vorticity function field, C, stream function field, Ψ, and wavelet base matrices Ω^{100} and Ω^{001}. The vorticity and stream function fields are assumed to be periodic wrap around square matrices, and $\hat{C}(p,q)$ and $\hat{\Psi}(p,q)$ are $m \times m$ matrices each, for an odd integer m, with entries from C and Ψ centered at the (p,q) element. Then the evaluation of (p,q) element of wavelet-Galerkin operator Θ is expressed as

$$\begin{aligned}
\Theta(p,q) &= \sum_{j=1}^{m}\sum_{k=1}^{m} \hat{H}(p,q)_{(j,k)} \\
&= \sum_{j=1}^{m}\sum_{k=1}^{m} \Big\{ \Big[\Omega^{100}\hat{C}(p,q) \; .* \; \hat{\Psi}(p,q)\Omega^{001} \Big] \\
&\quad - \Big[\Omega^{001}\hat{C}(p,q) \; .* \; \hat{\Psi}(p,q)\Omega^{100} \Big] \Big\}_{(i,j)}
\end{aligned} \tag{4.58}$$

and .* is the element-by-element product of two matrices. Fast algorithm for computation of Θ is based on a recursion relating $\hat{H}(p,q)$ to $\hat{H}(p-1,q)$ and $\hat{H}(p,q-1)$. Optimization in Jacobian is based on observations such as, if we know $\Omega^{100}\hat{C}(p,q)$, then $\Omega^{100}\hat{C}(p,q+1) = \Omega^{100}\left\{\hat{C}(p,q)\hat{D} + \hat{S}(p,q+1)\right\}$ where $\hat{D}$ is the matrix that shifts the columns to the left by one and assigns the null column to the last one and $\hat{S}(p,q+1)$ is the matrix with null columns except the last one which is the last column of $\hat{C}(p,q+1)$. This process reduces the evaluation of $\Omega^{100}\hat{C}(p,q+1)$ to a sequence of shifts and one matrix-vector multiplication. Similarly, $\hat{\Psi}(p,q)\Omega^{001}$ can be reduced to a sequence of matrix-vector multiplications and shifts. The module Helmholtz performs two-dimensional filtering of vorticity function with the Laplacian matrix, Δ^{-1}, via discrete Fourier transform methods.

4.3 Switching Between Data-Partitions

When an algorithm is to be designed in row-division for a data that is already partitioned in mesh-division, it is necessary to apply inverse mesh-division data-partitioning operation (see equation (3.28)) followed by forward column-division data-partitioning operation (see equation (3.24)): $\mathbf{P}_C\ \mathbf{P}_M^{-1} = \mathbf{P}_M^{-1}$ because, $\mathbf{P}_C$ is an identity matrix. Similarly, when the output data of an algorithm is in column-division but the required partition is mesh-division, then inverse column-division data-partitioning operation (see equation (3.27)) should be followed by forward mesh-division data-partitioning operation: $\mathbf{P}_M\ \mathbf{P}_C^{-1} = \mathbf{P}_M$ because, $\mathbf{P}_C^{-1}$ is an identity matrix. Assume that data is a two-dimensional array of size $M \times N$, and its mesh-division partition is performed on $k_s \times k_s$ grid, where $k_s^2 = k$.

4.3.1 2D-FFT from Mesh-Division via Column-Division: Algorithm-1

Starting at the equation similar to equation (3.53) for computing 2D-FFT using mesh-division data-partitioning (substitute $\mathbf{P}_M$ in place of $\mathbf{P}_R$ in equation (3.53)),

we derive complexity of $\mathbf{P}_M^{-1}$ as follows:

$$\mathbf{P}_M^{-1} = \left[\mathbf{I}_{k_s} \otimes P(N, N/k_s) \otimes \mathbf{I}_{M/k_s}\right]$$

by equation (3.28) and by theorem 2.7

$$\mathbf{P}_M^{-1} = \underbrace{\left[\mathbf{I}_k \otimes P(N/k_s, N/k) \otimes \mathbf{I}_{M/k_s}\right]}_{\mathcal{Z}_2} \underbrace{\left[\mathbf{I}_{k_s} \otimes P(k, k_s) \otimes \mathbf{I}_{MN/k_s^3}\right]}_{\mathcal{Z}_1} \tag{4.59}$$

Hence complexity of $\mathbf{P}_M^{-1}$ involves two stages. The first stage, $\mathcal{Z}_1$, involves $(k_s - 1)$ number of communications with respect to each processor. This is nothing but transpose algorithm within k_s-processors that belong to a column of processors. Similar transpose algorithms are performed simultaneously at k_s number of columns of processors where each column of processors consists of k_s processors. The second stage, $\mathcal{Z}_2$, is a local vector-stride data-shuffling.

From the above discussion on $\mathbf{P}_M^{-1}$, it can be easily found that $\mathbf{P}_M$ for mesh-division would also have same complexity (use $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$ and theorem 2.5 to equation (4.59). Hence $[\mathbf{P}_M P(MN, N)]$ at the output would require $(k + k_s - 2)$ communications in an unoptimized version because $P(MN, N)$ requires (refer Section 3.2) $(k - 1)$ stages of message-passing. Complexity of $P(MN, M)$ stays the same as the complexity of transpose algorithm for column-division partition as shown in equation (3.41). Optimization of communication at the output can be performed according to the following derivation:

$$\begin{aligned}
&\mathbf{P}_M P(MN, N) \\
&\quad = \left[\mathbf{I}_{k_s} \otimes P(N, k_s) \otimes \mathbf{I}_{M/k_s}\right] \left[P(Nk_s, N) \otimes \mathbf{I}_{M/k_s}\right] \\
&\qquad [\mathbf{I}_{k_s} \otimes P(MN/k_s, N)] \\
&\text{by theorem 2.8 and Definition 3.3} \\
&\quad = \underbrace{\left[P(k, k_s) \otimes \mathbf{I}_{MN/k}\right]}_{\mathcal{Z}_{11}} [\mathbf{I}_{k_s} \otimes P(MN/k_s, N)] \\
&\quad = \mathcal{Z}_{11} \left[\mathbf{I}_{k_s} \otimes P(Nk_s, N) \otimes \mathbf{I}_{M/k}\right] \underbrace{[\mathbf{I}_k \otimes P(MN/k, N)]}_{\mathcal{Z}_8}
\end{aligned}$$

$$= \mathcal{Z}_{11} \underbrace{\left[\mathbf{I}_k \otimes P(N, N/k_s) \otimes \mathbf{I}_{M/k}\right]}_{\mathcal{Z}_{10}} \underbrace{\left[\mathbf{I}_{k_s} \otimes P(k, k_s) \otimes \mathbf{I}_{MN/k_s^3}\right]}_{\mathcal{Z}_9} \mathcal{Z}_8 \qquad (4.60)$$

From first stage of transpose algorithm derived for mesh-division partition in Section 3.2, we know that $\mathcal{Z}_{11}$ represents one single communication. Also, $\mathcal{Z}_9$ is a transpose algorithm similar to the one seen in $\mathcal{Z}_1$ that requires $(k_s - 1)$ communication calls.

Hence, the total number of communications required for column-division are reduced from $(2k - 2)$ to $(k + 2k_s - 2)$ in mesh-division FFT algorithm. The final algorithm can be written as:

$$\begin{aligned}
\hat{\mathbf{y}} &= [\mathbf{P}_M P(MN, N)] \underbrace{[\mathbf{I}_M \otimes \mathbf{F}_N]}_{\mathcal{Z}_7} \underbrace{[P(MN, M)]}_{\mathcal{Z}_6\mathcal{Z}_5\mathcal{Z}_4} \underbrace{[\mathbf{I}_N \otimes \mathbf{F}_M]}_{\mathcal{Z}_3} \left[\mathbf{P}_M^{-1}\right] \hat{\mathbf{x}} \\
&= [\mathcal{Z}_{11}\mathcal{Z}_{10}\mathcal{Z}_9\mathcal{Z}_8]\,[\mathcal{Z}_7]\,[\mathcal{Z}_6\mathcal{Z}_5\mathcal{Z}_4]\,[\mathcal{Z}_3]\,[\mathcal{Z}_2\mathcal{Z}_1]\,\hat{\mathbf{x}} \\
&= \underbrace{\mathcal{Z}_{11}}_{\text{global3}}\ \underbrace{\mathcal{Z}_{10}}_{\text{v}}\ \underbrace{\mathcal{Z}_9}_{\text{global1}}\ \underbrace{\mathcal{Z}_8}_{\text{s}}\ \underbrace{\mathcal{Z}_7}_{\text{afft}}\ \underbrace{\mathcal{Z}_6}_{\text{s}}\ \underbrace{\mathcal{Z}_5}_{\text{global23}}\ \underbrace{\mathcal{Z}_4}_{\text{v}}\ \underbrace{\mathcal{Z}_3}_{\text{afft}}\ \underbrace{\mathcal{Z}_2}_{\text{v}}\ \underbrace{\mathcal{Z}_1}_{\text{global1}}\ \hat{\mathbf{x}}, \qquad (4.61)
\end{aligned}$$

where modules $\mathcal{Z}_4, \mathcal{Z}_5$, and $\mathcal{Z}_6$ are explained as block transpose algorithm in Section 3.2.

Module Description:

s, v: Local permutations: Simple and vector-stride permutations, respectively.

global1: $(k_s - 1)$ number of inter-node communications each of size $(MN/k_s^3) = k_s(MN/k^2)$. All these communications are one-to-one.

global23: $(k - 1)$ number of message passings, each of size (MN/k^2). All these communications are one-to-one.

global3: One node-to-node communication of size (MN/k) of type one-to-one.

afft: Routine to compute a sequence of one-dimensional FFTs.

4.3.2 2D-FFT from Mesh-Division via Column-Division: Algorithm-2

This method differs from algorithm-1 in the way we restructure at the output. So, we present here a different decomposition of last stage, $[\mathbf{P}_M P(MN, N)]$, of algorithm-1, First we derive a variant for $P(MN, N)$.

$$\begin{aligned} P(MN,N) &= [\mathbf{I}_{k_s} \otimes P(MN/k_s, N/k_s)] \left[P(Mk_s, k_s) \otimes \mathbf{I}_{N/k_s}\right] \\ &\text{by theorem 2.7} \\ &= \left[\mathbf{I}_{k_s} \otimes \left(P(N, N/k_s) \otimes \mathbf{I}_{M/k_s}\right) (\mathbf{I}_{k_s} \otimes P(MN/k, N/k_s))\right] \\ &\quad \left[P(Mk_s, k_s) \otimes \mathbf{I}_{N/k_s}\right] \\ &\text{by theorem 2.8 on} P(MN/k_s, N/k_s) \\ &= \mathbf{P}_M^{-1} [\mathbf{I}_k \otimes P(MN/k, N/k_s)] \left[P(Mk_s, k_s) \otimes \mathbf{I}_{N/k_s}\right] \\ &\text{by equation (3.28)} \end{aligned} \tag{4.62}$$

Hence,

$$\begin{aligned} \mathbf{P}_M P(MN,N) &= [\mathbf{I}_k \otimes P(MN/k, N/k_s)] \left[P(Mk_s, k_s) \otimes \mathbf{I}_{N/k_s}\right] \\ &= \underbrace{[\mathbf{I}_k \otimes P(MN/k, N/k_s)]}_{\mathcal{Z}_{10}} \underbrace{\left[P(k_s^3, k_s) \otimes \mathbf{I}_{MN/k_s^3}\right]}_{\mathcal{Z}_9} \\ &\quad \underbrace{\left[\mathbf{I}_k \otimes P(M/k_s, k_s) \otimes \mathbf{I}_{N/k_s}\right]}_{\mathcal{Z}_8} \end{aligned} \tag{4.63}$$

by theorem 2.8 (4.64)

Therefore, final implementation becomes:

$$\hat{\mathbf{y}} = \underbrace{\mathcal{Z}_{10}}_{\text{S}} \underbrace{\mathcal{Z}_9}_{\text{global4}} \underbrace{\mathcal{Z}_8}_{\text{V}} \underbrace{\mathcal{Z}_7}_{\text{afft}} \underbrace{\mathcal{Z}_6}_{\text{S}} \underbrace{\mathcal{Z}_5}_{\text{global23}} \underbrace{\mathcal{Z}_4}_{\text{V}} \underbrace{\mathcal{Z}_3}_{\text{afft}} \underbrace{\mathcal{Z}_2}_{\text{V}} \underbrace{\mathcal{Z}_1}_{\text{global1}} \hat{\mathbf{x}}, \tag{4.65}$$

where modules $\mathcal{Z}_1$ to $\mathcal{Z}_7$ are explained in previous section. Once again we reduced total communication cost from $(2k-1)$ to $(k+k_s-3)$, eliminating the one large and final communication in algorithm-1.

Problem Size	Nodes	Intel (*msecs*)	Interface (*msecs*)	Algorithm-1 (*msecs*)	Algorithm-2 (*msecs*)
32 × 32	4	0.06054	0.13752	0.12409	0.08476
	16	0.12427	0.25118	0.20137	0.13195
64 × 64	4	0.15091	0.31761	0.28070	0.23038
	16	0.13451	0.26424	0.23804	0.17571
	64	0.48014	0.72160	0.53387	0.39570
128 × 128	4	0.50754	0.96545	0.86153	0.76560
	16	0.24929	0.44145	0.42941	0.33604
	64	0.49421	0.76185	0.58775	0.43177
256 × 256	4	1.94816	3.43353	3.17574	2.91836
	16	0.60610	1.13566	1.15002	1.00119
	64	0.57530	0.94583	0.82859	0.64410
	256	1.96009	2.73886	1.66710	1.54402
512 × 512	4	8.58407	14.55625	13.08064	12.30499
	16	2.37530	4.07935	4.16807	3.81806
	64	1.09181	2.17609	1.92430	1.63670
	256	2.54740	2.90163	2.29605	1.96358

Table 4.5: Two-dimensional double-precision complex FFT implementation results for (1) $iPSC/860$ library code, (2) Interface routines appended at input and output, (3) Algorithm-1, and (4) Algorithm-2.

Module Description:

global4: $(k_s - 1)$ number of node-to-node communications each of size $(MN/k_s^3) = k_s(MN/k^2)$. Communications in **global1**, and **global23** are one-to-one implying two nodes form as a pair and swap contents between them. Types of communications in this module involve more than two nodes. For example, on a 16-node partition, communications at a stage are $0 \rightsquigarrow 4 \rightsquigarrow 5 \rightsquigarrow 9 \rightsquigarrow 10 \rightsquigarrow 14 \rightsquigarrow 15 \rightsquigarrow 3 \rightsquigarrow 0$, and $1 \rightsquigarrow 8 \rightsquigarrow 6 \rightsquigarrow 13 \rightsquigarrow 11 \rightsquigarrow 2 \rightsquigarrow 12 \rightsquigarrow 7 \rightsquigarrow 1$. Ideally, timings for such communications involving more than two nodes are not different from one-to-one communications, but depend upon the communications system that is present in the machine.

4.4 Effect of Varying Data Structures on Overall Performance: Results and Conclusion

Figures 4.3 and 4.4 show the contour plots of the vorticity functions obtained through the computational procedure shown in Figure 4.2 for various time steps. Each time step is initiated to 25 msecs and computation is carried out with tolerance of 10^{-4}. Performance results of FFT algorithms presented in Sections 4.3.1 and 4.3.2 versus the existing parallel FFT algorithm are presented in Table 4.5 for various sizes of data and machines. Third column represents the timings of FFT for row-division partition available in Intel's library while fourth column represents timings for FFT interfaced in unoptimized version. Columns 5 and 6 represent timings for algorithms-1 and 2 derived in Sections 4.3.1 and 4.3.2. It can be seen that algorithms-1 and 2 perform better than unoptimized version as expected. Moreover, for the case of 256-processor implementations, it is observed that these algorithms perform even better than FFT for row-division. This motivated us to derive another variant of FFT which is presented in the next chapter by eliminating block-transpose algorithm within all the processors.

Overall effect of the Jacobian computations are presented for row and mesh-division in second and third columns of Table 4.6. It is to be observed that speed-up is more linear as machine size increases in case of mesh-division data-partitioning compared to row-division data-partitioning. This enabled improvement of the overall performance of the application. Columns 4, 5, and 6 present timing results for evaluating two-dimensional filtering in Helmholtz module using different FFT algorithms. Improvements of PDE solution using FFT algorithms-1 and 2 that start with mesh-division partitioned data are presented in last two columns of the Table 4.6. Shown results are averaged over entire computational procedure and up to 43.61% reduction in time is achieved. Hence, we conclude that the choice of data-partition and efficient manipulation of our algebraic definitions for data-partition indeed helps to improve the overall performance of an application.

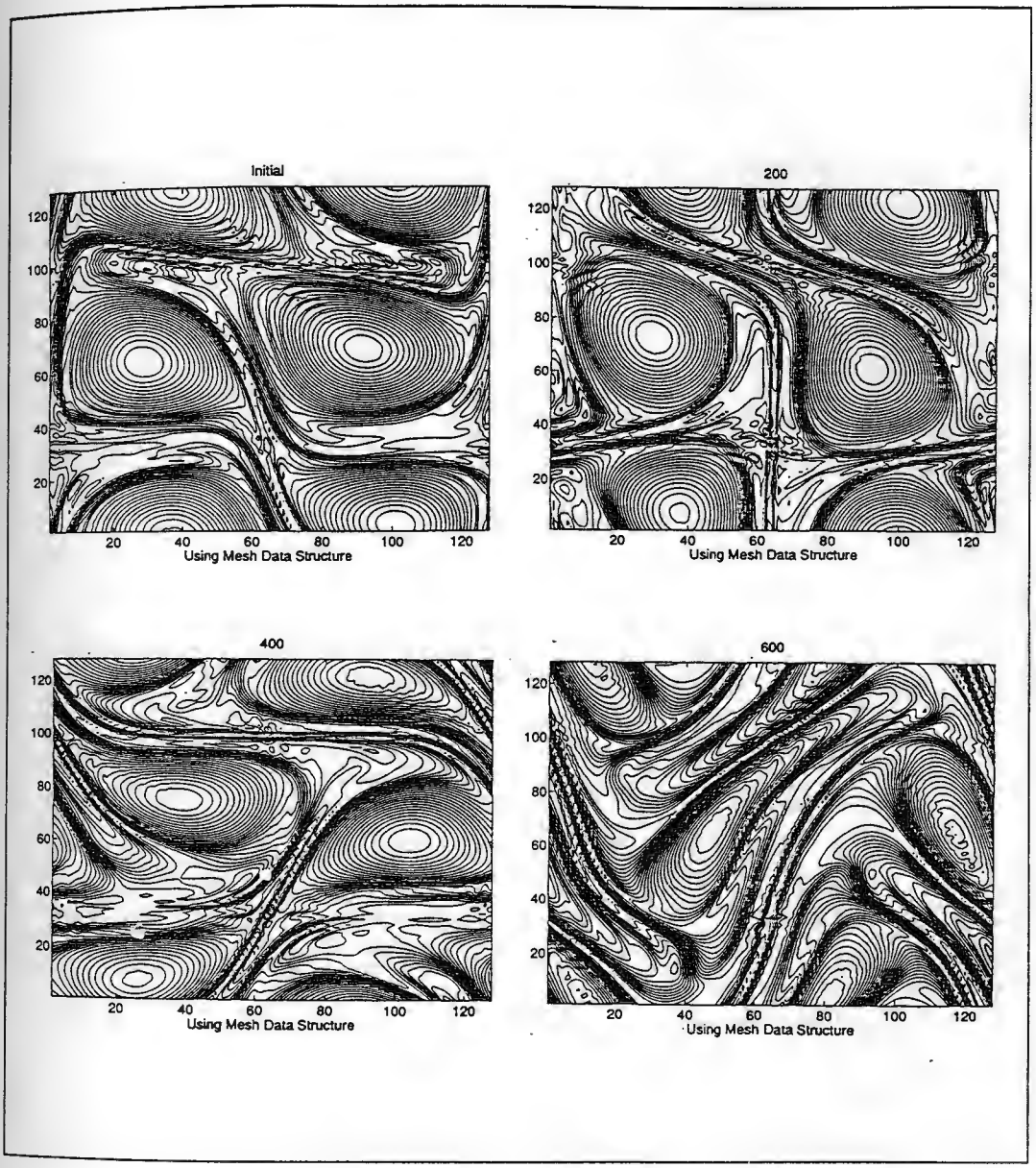


Figure 4.3: Contour plots of the Initial vorticity function and for time steps 200, 400, and 600.

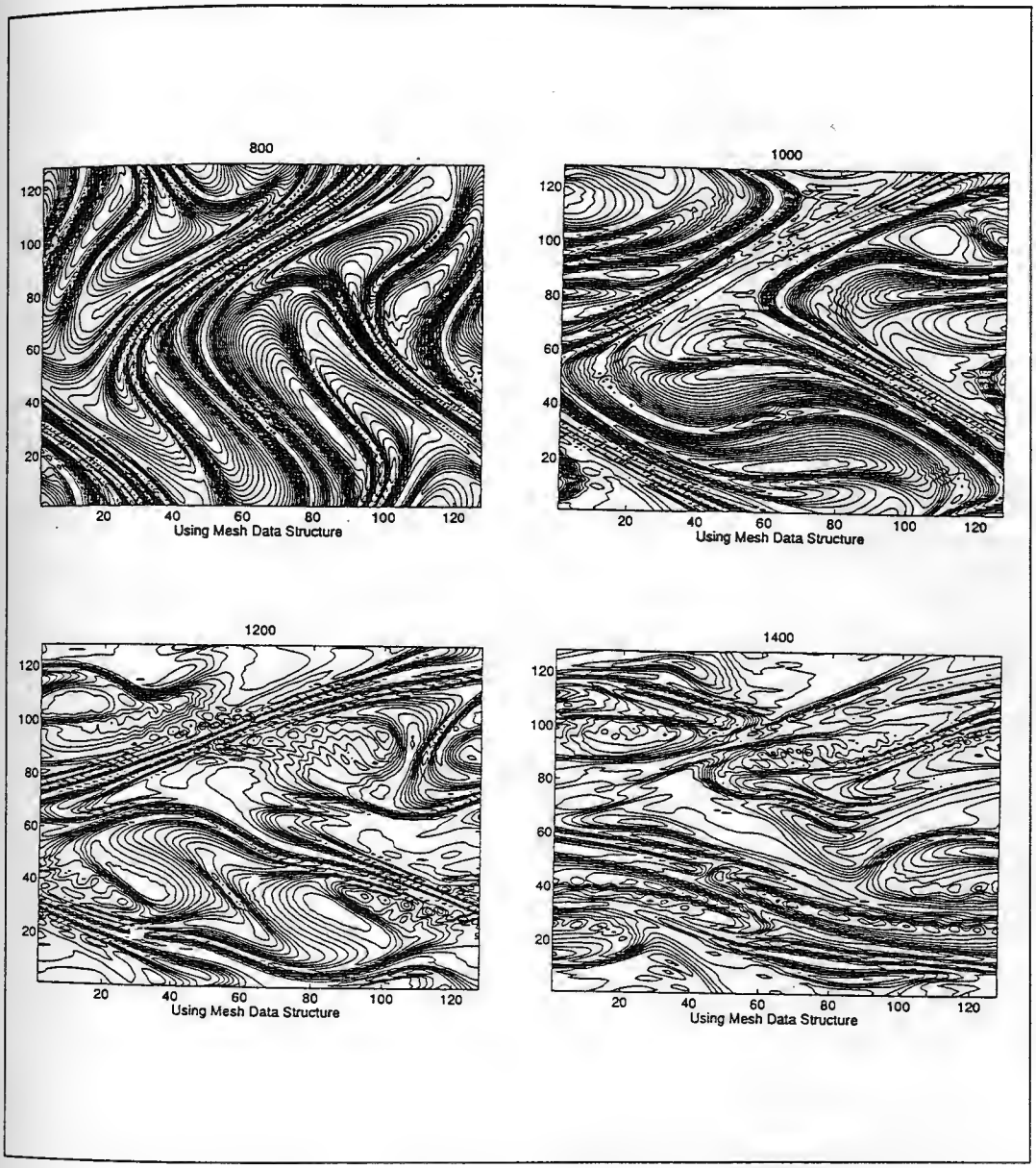


Figure 4.4: Contour plots of the vorticity functions for time steps 800, 1000, 1200, and 1400.

Nodes	Jacobian		Helmholtz			Total		
	row-D	Mesh	row-D	Mesh1	Mesh2	row-D	Mesh1	Mesh2
4	2.8317	2.7939	0.11216	0.18218	0.16298	2.9438	2.9761	2.9568
16	0.8128	0.7310	0.06094	0.09950	0.07688	0.8738	0.8305	0.8079
64	0.3095	0.1996	0.10510	0.12022	0.08916	0.4146	0.3198	0.2887

Table 4.6: Timing results for 128×128 size vorticity computations

Chapter 5

A New Approach for FFT Algorithm with Mesh-Division

5.1 Introduction

This chapter presents a new and optimal parallel implementation of multidimensional fast Fourier transform algorithm on distributed memory multiprocessors based on variations in communication strategies. Its optimality is obtained by minimizing the number of necessary message-passings at the cost of increase in message length. This distinctive feature of the new algorithm effectively utilizes the important architectural property of most today's distributed memory multiprocessors – wormhole routing for interprocessor communications. By using the algebra of stride permutations and tensor products as a mathematical tool, we are able to derive and formulate an efficient data-partition and communication scheme that reduces communication cost from $O(k)$ required for the best known FFT to $O(\sqrt{k})$ on an k-processor machine. The data-partition considered here (mesh) is natural and efficient for solving discretized boundary value problems such as partial differential equations and finite element analysis discussed in Chapter 4. To evaluate the actual performance of our new algorithm in comparison with other existing parallel FFT algorithms, we have carried out implementation experiments on the Intel's Touchstone Delta. Experimental results show that our algorithm is highly efficient and runs up to 6 times faster than the existing algorithm on a 128 or 256 nodes machine for complex data

size ranging from 16K to 1M points. What is more interesting is that the finer the parallelism is, the better the new algorithm performs than the existing [35] ones presented in Section 3.3.

Consider distributed memory multiprocessor environment where communication is done through message-passing. Traditionally, research in this field has concentrated on localizing communications so that data messages are passed only between neighboring processors, processors that are directly connected by a physical link. The reason for such efforts is that most distributed shared memory multiprocessors are not fully connected like hypercube, mesh, or rings. Each processor is connected only to a few neighboring processors and communication between nonneighboring processors has to go through one or several intermediate nodes. It was believed that an algorithm that allows communication to be done only among neighboring processors will minimize communication cost. However, in most today's message-passing multiprocessors, wormhole routing techniques are used to route messages among processors [12, 53]. The pipelining nature of wormhole routing makes the network latency insensitive to path length. In other words, communication latency is virtually independent of the physical distance between two communicating processors. Therefore, neighboring communication is no longer the key factor in an algorithm design and the traditional way of designing parallel algorithms may no longer give the optimal performance. Our objective here is to show that much more performance gains are possible by exploiting the new architectural features such as wormhole routing techniques in multiprocessor systems.

Using the algebra of stride permutations and tensor products as mathematical tools, we have seen that one can easily manipulate the communication structure of an algorithm and derive a structure best tailored to the underlying architecture and develop an optimal communication structure. Such association between tensor notation and algorithm design provides a new way of understanding and developing efficient parallel algorithms. Our new parallel FFT algorithm that arises from such mathematical manipulation minimizes the total number of messages that have to be

passed among processors in the course of the FFT computation. This reduction of the total number of messages comes at the expense of increase in the length of each message. Since the network setup time in wormhole routing plays more significant role in latency than message length due to pipelining, our algorithm shows significant better performance than existing parallel algorithms [35].

Using equation (1.1) and with the explanation in Section 3.3, one can estimate the total communication cost of the existing algorithm [35] on a k-node machine. Fourier transforming a two-dimensional data of size $kN_1 \times kN_2$ involves inter-processor communication cost given by

$$t_{Old} = 2(k-1)\left[t_s + N_1 N_2 t_e\right] \tag{5.66}$$

The chapter is organized as follows. In the following, we analyzed the performance bottlenecks of existing FFT algorithm to find where improvements are possible. Section 5.2 presents our new algorithm using tensor notations, and proves its validity. Experiments and performance measurements will be presented in Section 5.3. Finally, Section 5.4 concludes our approach.

5.2 New Approach

Having analyzed the communication cost of the existing FFT algorithm for row-division partition, we focus our effort at reducing the cost expressed in equation (5.66) [54]. It is clear that the parameter t_s is a major factor contributing to the total cost compared to the parameter t_e. Our main objective is to minimize the role of network setup time in communication overhead. The main idea is similar to the classical divide-and-conquer strategy in algorithm designs. Suppose that the size of the machine can be represented by two factors, i.e., $k = k_1 k_2$. With proper data-partitioning and allocation, we reduced the network setup time from $O(k_1 k_2)$ to $O(k_1 + k_2)$ with an increase of a constant factor for the coefficient of t_e (see equation (5.77) in the next section).

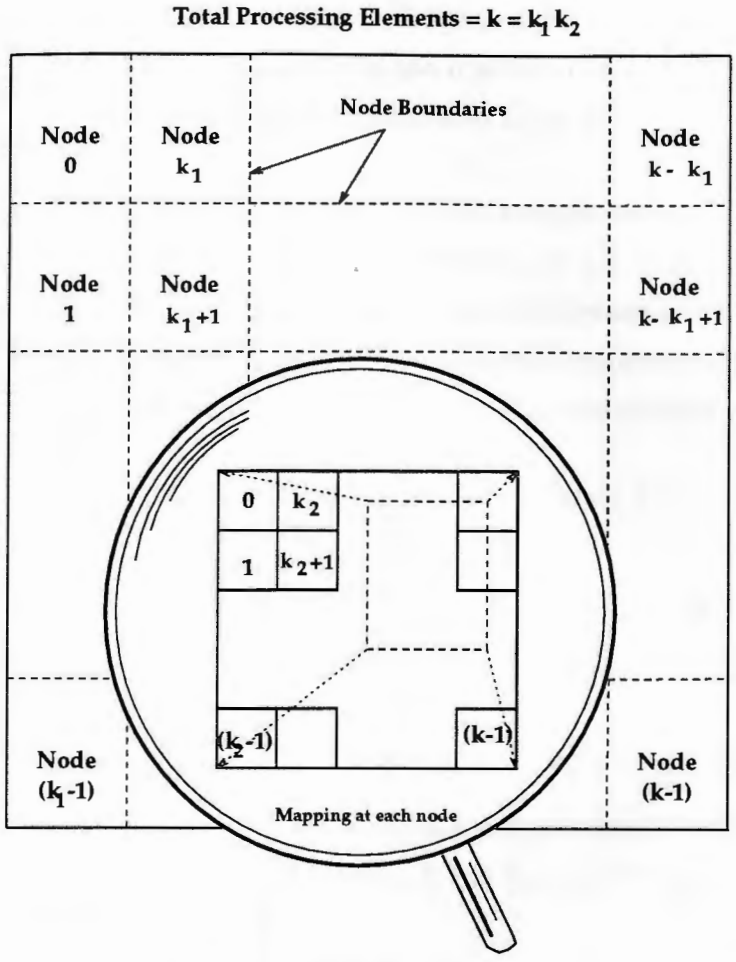


Figure 5.5: Mapping of 2-D array $f(x, y)$ onto 6-D array.

Let the input data to be transformed be a two-dimensional array with size $kN_1 \times kN_2$. If this computation is to be carried out on k-processor machine, then each processor would be assigned with kN_1N_2 length sub-vectors. Such sub-vectors are obtained by tiling the two-dimensional array into $k_1 \times k_2$ blocks of size $k_2N_1 \times k_1N_2$.

These $k_1 \times k_2$ blocks of size $k_2N_1 \times k_1N_2$ are shown with dotted lines in Figure 5.5. We then allocate each sub-block to a processor. This process is typical in finite element analysis where finer the grid, higher the complexity of computation. With $k_1 \times k_2$ blocks, we imagine associated processors are being arranged in k_2 columns,

each column consisting of k_1 processors. These processors are numbered in anti-lexicographic manner, that is, processors in the first column are numbered from 0 to $(k_1 - 1)$, those in the second column are numbered from k_1 to $(2k_1 - 1)$, and so on.

Similar to *Vect* operation in Section 2.2, we form a single vector $\mathbf{x}$ out of the input matrix by placing column-$(i+1)$ down the column-i, $1 \leq i \leq (kN_2 - 1)$ (column-major). Then, shuffled vector $\hat{\mathbf{x}}$ built with sub-vectors are assigned to processors 0 to $(k-1)$. The tensor product and stride permutation representing all the above data-partitioning is seen in equation (3.25) that can be represented as:

$$\hat{\mathbf{x}} = \underbrace{\left[\mathbf{I}_{k_2} \otimes P(k_1^2 N_2, k_1) \otimes \mathbf{I}_{k_2 N_1}\right]}_{\mathbf{P}_M(kN_1, kN_2, k_1, k_2)} \mathbf{x} \tag{5.67}$$

$$= \begin{bmatrix} P(k_1^2 N_2, k_1) \otimes \mathbf{I}_{k_2 N_1} & & \emptyset \\ & P(k_1^2 N_2, k_1) \otimes \mathbf{I}_{k_2 N_1} & \\ & & \ddots \\ \emptyset & & P(k_1^2 N_2, k_1) \otimes \mathbf{I}_{k_2 N_1} \end{bmatrix} \mathbf{x}$$

$$\begin{bmatrix} \hat{\mathbf{x}}(0 : J-1) \\ \hat{\mathbf{x}}(J : 2J-1) \\ \vdots \\ \hat{\mathbf{x}}((k_2-1)J : k_2 J - 1) \end{bmatrix} = \begin{bmatrix} (P(k_1^2 N_2, k_1) \otimes \mathbf{I}_{k_2 N_1})\mathbf{x}(0 : J-1) \\ (P(k_1^2 N_2, k_1) \otimes \mathbf{I}_{k_2 N_1})\mathbf{x}(J : 2J-1) \\ \vdots \\ (P(k_1^2 N_2, k_1) \otimes \mathbf{I}_{k_2 N_1})\mathbf{x}((k_2-1)J : k_2 J - 1) \end{bmatrix}$$

where $J = k_2 k_1^2 N_1 N_2$ and $\mathbf{x}(x_1 : x_2)$ represents sub-vector of $\mathbf{f}$ formed with elements with subscripts from x_1 to x_2. In the above matrix representation, each row denotes the operation that is being performed on an entire column of k_1 processors. Recall that a tensor product with post identity matrix, $\mathbf{I}_{k_2 N_1}$, represents operations on vectors of length $k_2 N_1$, and with the operational matrix being $P(k_1^2 N_2, k_1)$, it represents picking up vectors of length $k_2 N_1$ with stride k_1. Tensor product with prior identity matrix, $\mathbf{I}_{k_2}$ in equation (5.67), represents that similar operations are to be performed on each of the k_2 columns of processors. Extension of such a data-partition and allocation scheme for higher dimensional problems can be done in a similar way although the geometrical representation would be more complicated.

This is clear from zoomed part of Figure 5.5 in which we detailed the required partition for implementation with another grid of size $k_2 \times k_1$ at each node for representing two-dimensional algorithm making a six-dimensional indexing instead of our tensor product formulation presented below. Each block in this grid would consist $N_1 \times N_2$ elements.

With the data allocation made in above for computing two-dimensional DFT, the computation proceeds in stages that are explained in the following. Stride permutations and tensor product are aid to clearly visualize the complexity and feasibility of execution on parallel machines.

Rearrange I:

$$\mathbf{f_2} = \left[\mathbf{I}_{k_2} \otimes P(k_1^2, k_1) \otimes \mathbf{I}_{k_2 N_1 N_2}\right] \hat{\mathbf{x}} \tag{5.68}$$

This stage involves simultaneous message-passing among all the processors that belong to the same column. In fact, any rearrangement that is not preceded by $\mathbf{I}_k$ would result in inter-processor communications. However, $\mathbf{I}_{k_2}$ preceding the above expression indicating that k_2 columns of processors are doing intra-column message-passings in parallel. Actual implementation involves message-passing from a node *sender* to a different node *receiver*.

1. Node *sender* calling a procedure that sends a vector to node *receiver* with parameters (a) vector's name, (b) size, (c) address of the node to which data should be directed to, which is *receiver* in this case, and (d) a user defined message number that should be same for all the global communications being performed at that time.

2. Node *receiver* employing a routine for an asynchronous receive that initiates the receipt of a message from a process.

3. Node *receiver* utilizing a message wait routine to block further execution of any instructions that are dependent upon data in transit until the transfer is complete.

Hence, this stage of global data shuffling involves a complexity of $(k_1 - 1)$ number of messages to be passed by each processor, each message being of length $k_2 N_1 N_2$.

Rearrange II:

$$\mathbf{f_3} = [\mathbf{I}_k \otimes P(k_1 N_2, N_2) \otimes \mathbf{I}_{k_2 N_1}]\, \mathbf{f_2} \tag{5.69}$$

Compute I:

$$\mathbf{f_4} = [\mathbf{I}_{kN_2} \otimes \mathbf{F}_{kN_1}]\, \mathbf{f_3} \tag{5.70}$$

Rearrange III:

$$\mathbf{f_5} = [\mathbf{I}_k \otimes P(k_1 N_2, k_1) \otimes \mathbf{I}_{k_2 N_1}]\, \mathbf{f_4} \tag{5.71}$$

In Rearrange II, the tensor products have densely packed knowledge about the implementation aspects. First of all, the occurrence of the identity matrix I_k on the left indicates parallel computations. Therefore, operation at a node is independent of any other node. The actual operations performed in parallel are data shuffling to obtain a complete column of the input matrix in order at each processor. As a result of this rearrangement, processor-0 contains the first N_2 columns in the natural order, processor-1 contains the next N_2 columns in the natural order, and so forth.

After N_2 columns are obtained at each processor, Compute I performs N_2 number of kN_1-point one-dimensional Fourier transforms on columns, using an efficient one-dimensional transform routine called `cfft1d` developed by Kuck & Associates Inc., for complex numbers. It is evident from theorems 2.9 and 2.5 in Section 2.5 that Rearrange III is the inverse operation of Rearrange II. It scatters back the transformed vectors of length $k_2 N_1$ with stride k_1 to prepare for the global communications in the next stage. This step is once again independent of the data at other nodes and can be executed in parallel.

Rearrange IV:

$$\mathbf{f_6} = \left[\mathbf{I}_{k_2} \otimes P(k_1^2, k_1) \otimes \mathbf{I}_{k_2 N_1 N_2}\right] \mathbf{f_5} \tag{5.72}$$

This stage is exactly the same as Rearrange I that has a complexity of $(k_1 - 1)$ global communications, each message has the length $k_2 N_1 N_2$.

Rearrange V:

$$\mathbf{f_7} = [\mathbf{I}_k \otimes P(kN_1N_2, k_2N_1)]\,\mathbf{f_6} \tag{5.73}$$

With I_k preceding this operation, this is nothing but local transpose of the matrices of size $k_2N_1 \times k_1N_2$ at each processor. This transpose would prepare the data to perform similar stages as above on the next dimension.

Following six stages are identical to the above six stages except that they are being performed on the second dimension. However, just for the purpose of validation through out the algorithm, terms $[P(k, k_1) \otimes \mathbf{I}_{kN_1N_2}]$ and $[P(k, k_2) \otimes \mathbf{I}_{kN_1N_2}]$ are introduced, first before and then after the global communications in Rearrange VI and IX. However, they are not seen in actual implementation because destination processors are addressed with the necessary node numbers.

Rearrange VI: $\mathbf{f_8} = [\mathbf{I}_{k_2} \otimes P(k_2^2, k_2) \otimes \mathbf{I}_{k_1N_1N_2}]\,[P(k, k_1) \otimes \mathbf{I}_{kN_1N_2}]\,\mathbf{f_7}$.

Rearrange VII: $\mathbf{f_9} = [\mathbf{I}_k \otimes P(k_2N_1, N_1) \otimes \mathbf{I}_{k_1N_2}]\,\mathbf{f_8}$.

Compute II: $\mathbf{f_{10}} = [\mathbf{I}_k \otimes \mathbf{F}_{kN_2}]\,\mathbf{f_9}$.

Rearrange VIII: $\mathbf{f_{11}} = [\mathbf{I}_k \otimes P(k_2N, k_2) \otimes \mathbf{I}_{k_1N_2}]\,\mathbf{f_{10}}$.

Rearrange IX: $\mathbf{f_{12}} = [P(k, k_2) \otimes \mathbf{I}_{kN_1N_2}]\,[\mathbf{I}_{k_2} \otimes P(k_2^2, k_2) \otimes \mathbf{I}_{k_1N_1N_2}]\,\mathbf{f_{11}}$.

Rearrange X: $\hat{\mathbf{y}} = [\mathbf{I}_k \otimes P(kN_1N_2, k_1N_2)]\,\mathbf{f_{12}}$.

5.2.1 Proof

If we consider data $\hat{\mathbf{x}}$ of size $kN_1 \times kN_2$ arranged on a $k_1 \times k_2$ grid to be Fourier transformed to data $\hat{\mathbf{y}}$, then

$$\begin{aligned}
\hat{\mathbf{y}} &= \mathbf{P}_M(kN_1, kN_2, k_1, k_2)\,[\mathbf{F}_{kN_2} \otimes \mathbf{F}_{kN_1}] \\
&\quad \mathbf{P}_M^{-1}(kN_1, kN_2, k_1, k_2)\hat{\mathbf{x}} \\
&= \mathbf{P}_M(kN_1, kN_2, k_1, k_2)\,[\mathbf{F}_{kN_2} \otimes \mathbf{I}_{kN_1}] \\
&\quad [\mathbf{I}_{kN_2} \otimes \mathbf{F}_{kN_1}]\,\mathbf{P}_M^{-1}(kN_1, kN_2, k_1, k_2)\hat{\mathbf{x}} \\
&= \mathbf{P}_M(kN_1, kN_2, k_1, k_2)P(k^2N_1N_2, kN_2)\,[\mathbf{I}_{kN_1} \otimes \mathbf{F}_{kN_2}]
\end{aligned}$$

$$P(k^2N_1N_2, kN_1)\,[\mathbf{I}_{kN_2} \otimes \mathbf{F}_{kN_1}]\,\mathbf{P}_M^{-1}(kN_1, kN_2, k_1, k_2)\hat{\mathbf{x}}$$

Using equations (3.47) and (3.48) to expand matrix transposes $P(k^2N_1N_2, kN_2)$ and $P(k^2N_1N_2, kN_1)$, respectively,

$$\begin{aligned}
\hat{\mathbf{y}} \;=\; & \underbrace{[\mathbf{I}_k \otimes P(kN_1N_2, k_1N_2)]}_{\text{Rearrange X}}\,[P(k, k_2) \otimes \mathbf{I}_{kN_1N_2}] \\
& \mathbf{P}_M(kN_2, kN_1, k_2, k_1)\,[\mathbf{I}_{kN_1} \otimes \mathbf{F}_{kN_2}] \\
& \mathbf{P}_M^{-1}(kN_2, kN_1, k_2, k_1)\,[P(k, k_1) \otimes \mathbf{I}_{kN_1N_2}] \\
& \underbrace{[\mathbf{I}_k \otimes P(kN_1N_2, k_2N_1)]}_{\text{Rearrange V}}\,\mathbf{P}_M(kN_1, kN_2, k_1, k_2) \\
& [\mathbf{I}_{kN_1} \otimes \mathbf{F}_{kN_1}]\,\mathbf{P}_M^{-1}(kN_1, kN_2, k_1, k_2)\hat{\mathbf{x}} \qquad (5.74)
\end{aligned}$$

Counting steps from bottom to top in the above equation, steps 1–3 and 6–8 are dual and one can be obtained from the other by exchanging N_1 with N_2, and k_1 with k_2. Consider an operational matrix $\mathbf{A}$ that consists the stages Rearrange I, II, Compute I, Rearrange III, and IV explained in the above section. Then we will prove that steps 1–3 in above equation are equivalent to $\mathbf{A}$. The steps 6–8 can be proved in a similar fashion to be equivalent to respective stages in the other dimension.

$$\begin{aligned}
& \mathbf{P}_M^{-1}(kN_1, kN_2, k_1, k_2)\ \mathbf{A}\ \mathbf{P}_M(kN_1, kN_2, k_1, k_2) \\
= \; & \underbrace{\left[\mathbf{I}_{k_2} \otimes P(k_1^2N_2, k_1N_2) \otimes \mathbf{I}_{k_2N_1}\right]}_{\mathbf{P}_M^{-1}}\underbrace{\left[\mathbf{I}_{k_2} \otimes P(k_1^2, k_1) \otimes \mathbf{I}_{k_2N_1N_2}\right]}_{\text{Rearrange IV}} \\
& \underbrace{[\mathbf{I}_k \otimes P(k_1N_2, k_1) \otimes \mathbf{I}_{k_2N_1}]}_{\text{Rearrange III}}\underbrace{[\mathbf{I}_{kN_2} \otimes \mathbf{F}_{kN_1}]}_{\text{Compute I}} \\
& \underbrace{[\mathbf{I}_k \otimes P(k_1N_2, N_2) \otimes \mathbf{I}_{k_2N_1}]}_{\text{Rearrange II}}\underbrace{\left[\mathbf{I}_{k_2} \otimes P(k_1^2, k_1) \otimes \mathbf{I}_{k_2N_1N_2}\right]}_{\text{Rearrange I}} \\
& \underbrace{\left[\mathbf{I}_{k_2} \otimes P(k_1^2N_2, k_1) \otimes \mathbf{I}_{k_2N_1}\right]}_{\mathbf{P}_M} \\
= \; & \left(\mathbf{I}_{k_2} \otimes \left\{P(k_1^2N_2, k_1N_2\left[P(k_1^2, k_1) \otimes \mathbf{I}_{N_2}\right]\left[\mathbf{I}_{k_1} \otimes P(k_1N_2, k_1)\right]\right\} \otimes \mathbf{I}_{k_2N_1}\right) \\
& [\mathbf{I}_{kN_2} \otimes \mathbf{F}_{kN_1}]
\end{aligned}$$

$$
\begin{aligned}
& \left(\mathbf{I}_{k_2} \otimes \left\{[\mathbf{I}_{k_1} \otimes P(k_1 N_2, N_2)]\left[P(k_1^2, k_1) \otimes \mathbf{I}_{N_2}\right] P(k_1^2 N_2, k_1)\right\} \otimes \mathbf{I}_{k_2 N_1}\right) \\
= & \left[\mathbf{I}_{k_2} \otimes \mathbf{I}_{k_1^2 N_2} \otimes \mathbf{I}_{k_2 N_1}\right]\left[\mathbf{I}_{kN_2} \otimes \mathbf{F}_{kN_1}\right]\left[\mathbf{I}_{k_2} \otimes \mathbf{I}_{k_1^2 N_2} \otimes \mathbf{I}_{k_2 N_1}\right] \\
= & \left[\mathbf{I}_{kN_2} \otimes \mathbf{F}_{kN_1}\right] \qquad (5.75)
\end{aligned}
$$

Therefore,

$$
\mathbf{A} = \mathbf{P}_M(kN_1, kN_2, k_1, k_2)\left[\mathbf{I}_{kN_2} \otimes \mathbf{F}_{kN_1}\right]\mathbf{P}_M^{-1}(kN_1, kN_2, k_1, k_2) \qquad (5.76)
$$

Hence the proof.

5.3 Performance Evaluation and Comparison

In this section, we evaluate the performance of the new approach developed in the previous section. We will also compare its performance with the existing algorithm described in equation (5.66). It is clear that computational complexity is the same for both algorithms. We can estimate the inter-processor communication cost similar to equation (5.66) for the new algorithm. It is given by

$$
t_{New} = 2(k_1 - 1)[t_s + k_2 N_1 N_2 t_e] + 2(k_2 - 1)[t_s + k_1 N_1 N_2 t_e] \qquad (5.77)
$$

From equation (5.77), we can see that for our new approach to be efficient than the existing algorithm [35], the following must hold.

$$
\begin{aligned}
t_{New} &< t_{Old} \\
(kN_1)(kN_2) &< (t_s/t_e)(k)^2 \qquad (5.78) \\
\text{Data size} &< (t_s/t_e)(\text{Machine size})^2.
\end{aligned}
$$

Experiments to measure the actual performance of the two algorithms on the Delta machine have been carried out. The measurements are reported in Table 5.7. The results shown in this table are measured with a library routine called `dclock()` that returns a double precision number. Using this routine at the beginning and at the end of each of the algorithms, we obtained double precision time in milli-seconds. These timings are purely for execution of the FFT algorithm because processors

are not time-shared by multiple users. However, since each node would execute in a slightly different time due to the asynchronous communication network, we considered the maximum value of the times reported by all the nodes. Also, we have averaged timings over a set of one hundred experiments with forward and inverse two-dimensional transforms for each data size.

Performance of two different algorithms are reported by executing them on 128-node and 256-node machine-partitions. Various data sizes that we have tested are presented in the first column in Table 5.7. The second and third columns represent timings for existing and new approaches, respectively, on 128-node machine while fourth and fifth columns are for the cases of 256-node machine.

One can find a direct correlation between the theoretical estimation of performance in equation (5.77) and the results in Table 5.7. It can be seen that as the machine size increases performance of new approach increases. This is because the complexity of network startups in the existing algorithm is $O(k_1 k_2)$ while that of the new approach is $O(k_1 + k_2)$. In general, for an n-dimensional DFT computation on $(k_1 k_2 \ldots k_n)$-processor machines, order of network startups for existing algorithm would be $O(\Pi_{i=1}^{n} k_i)$ while that of new approach would be $O(\sum_{i=1}^{n} k_i)$. Therefore, the data-partitioning scheme and communication setup described in new approach would be especially useful in the problems with the combination of huge data size, large machines, and higher dimensionality.

5.4 Conclusion

In Section 5.2, we presented an approach for computing multidimensional DFT on distributed memory systems that effectively utilizes the fact that today's distributed memory systems use wormhole routing for interprocessor communications. An approach to extend the algorithm for three or more dimensional problems using stride permutation and tensor product matrices has been presented that facilitates finding

Data Size			128 nodes		256 nodes	
			Old	New	Old	New
M	$\times$	N	*(msecs)*	*(msecs)*	*(msecs)*	*(msecs)*
128	$\times$	128	120.117	27.727	N/A	31.711
256	$\times$	128	120.151	31.234	192.980	35.017
256	$\times$	256	121.681	34.165	245.634	39.499
512	$\times$	128	125.425	34.401	210.761	35.865
512	$\times$	256	129.847	44.944	254.412	44.948
1024	$\times$	128	128.236	44.883	227.441	43.225
512	$\times$	512	125.901	60.946	270.365	56.096
1024	$\times$	256	133.562	64.331	262.051	53.420
1024	$\times$	512	152.919	99.989	285.066	76.041
1024	$\times$	1024	211.274	177.306	294.038	119.288

Table 5.7: Implementation results of FFT using new approach on Intel's Touchstone Delta.

an efficient data-partitioning and network setup on distributed memory multiprocessors. Data-partitioning scheme is suitable and should be aimed at boundary value problems in fluid dynamics, finite element analysis etcetera. Results showed that our algorithm is more than six times as fast as the existing algorithm for certain cases. Moreover, higher the parallelism is, the better the performance of new algorithm will be. Given the fact that physical limits on memory exist at each processor, our new algorithm is a solution to today's large problems that involve multidimensional Fourier transform computations on massively parallel machines.

Chapter 6

Parallel Matrix Multiplication Algorithm For Rectangular Arrays

6.1 Introduction

Many applications have numerical solutions in which computational burden is reduced partly or fully to matrix operations. One of the most elementary operations involving matrices is multiplication of two matrices. However, since matrix multiplication requires substantially more data movements than most other operations, algorithms that address efficient data movements are crucial to their effective implementation on concurrent computers.

This chapter is organized as follows. Section 6.2 reviews an existing matrix multiplication algorithm that generates and accumulates partial results by moving multiplicands through a set of broadcasts and shifts. Section 6.3 considers the two extreme cases of the broadcast-and-shift multiplication algorithm arising from data decomposition strategies. These cases involve either only a set of broadcasts or only a set of shifts. We present a new approach in Section 6.4 that replaces broadcasts or shifts by matrix transpose. Identification of shortcomings in the two extreme cases of broadcast-and-shift algorithm and the fact that dot product of two vectors result in a single element is the motivation for this new approach. Then, to overcome the

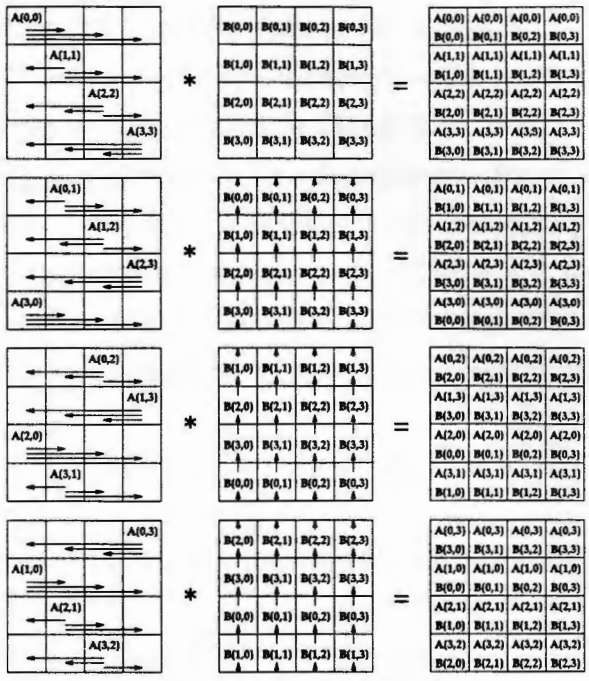


Figure 6.6: Broadcast-and-Shift Matrix Multiplication Algorithm on 16-processors

hurdles in memory requirement, we modified the algorithm for efficient data manipulation with the aid of block transpose algorithm seen in Chapter 3. Section 6.5 presents theoretical evaluation of communication costs of broadcast-and-shift algorithm versus new approach and timing results of their implementations on Intel's Paragon, Touchstone Delta, and *iPSC/860* which inferred that new approach is indeed efficient for rectangular arrays. Section 6.6 concludes the chapter.

6.2 Broadcast-and-Shift Matrix Multiplication Algorithm

This section reviews the broadcast-and-shift matrix multiplication algorithm that is presented in [25]. Related research can be found in [55, 56, 57]. Throughout this chapter, we consider computing $\mathbf{C}$, where

$$\mathbf{C} = \mathbf{A}\ \mathbf{B},$$

on concurrent processors. Let k be the number of processors in a distributed memory system. Assuming k to be a square of an integer, $k = k_s^2$, we can use square subblock decomposition (mesh-division) as shown in Figure 6.6. Then, multiplicand matrices $\mathbf{A}$ and $\mathbf{B}$ are distributed piecewise in two-dimensions. Resultant matrix $\mathbf{C}$ is also expected to be distributed in the same fashion for further processing steps. Denote a processor belonging to ith row and jth column by $P^{(i,j)}$, $0 \leq i, j \leq (k_s - 1)$. Similarly, denote the subblocks of matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ in processor $P^{(i,j)}$ by $\mathbf{A}^{(i,j)}$, $\mathbf{B}^{(i,j)}$, and $\mathbf{C}^{(i,j)}$, respectively. The multiplication with respect to blocks can be written as

$$\mathbf{C}^{(i,j)} = \sum_{l=0}^{(k_s-1)} \left(\mathbf{A}^{(i,l)} \, \mathbf{B}^{(l,j)} \right). \tag{6.79}$$

The above equation can be rewritten for implementation on a multiprocessor environment as:

$$\mathbf{C}^{(i,j)} = \sum_{l=0}^{(k_s-1)} \left(\mathbf{A}^{(i,l_1)} \, \mathbf{B}^{(l_1,j)} \right) \tag{6.80}$$

where $l_1 \equiv (i + l) \pmod{k_s}$, and k_s is the number of processors in a row. Equation (6.80) represents operations to be performed at processor $P^{(i,j)}$. These operations are divided into k_s stages of operations, one stage for each l, $0 \leq l \leq (k_s - 1)$, in the summation. Consider dividing each stage into two tasks: (a) task that involves message-passings to obtain $\mathbf{A}^{(i,l_1)}$ and $\mathbf{B}^{(l_1,j)}$ at processor $P^{(i,j)}$ and (b) task that involves computation of the product $\mathbf{A}^{(i,l_1)} \, \mathbf{B}^{(l_1,j)}$ at processor $P^{(i,j)}$ and accumulation of the result to $\mathbf{C}^{(i,j)}$. To compute equation (6.80) in $P^{(i,j)}$ for $j = 1 \cdots k_s$, all the k_s processors belonging to the same row, with same index i, should obtain $\mathbf{A}^{(i,l_1)}$, for all values of l_1 where $l_1 \equiv (i + l) \pmod{k_s}$, and $0 \leq l \leq (k_s - 1)$. This is done by broadcasting from processor $P^{(i,l_1)}$ to all the processors in same row for each stage. To obtain $\mathbf{B}^{(l_1,j)}$ at processor $P^{(i,j)}$ for each l, one needs to shift subblocks of $\mathbf{B}$ up after each stage, as shown in Figure 6.6. Once subblocks $\mathbf{A}^{(i,l_1)}$ and $\mathbf{B}^{(l_1,j)}$ are obtained at processor $P^{(i,l_1)}$, they are multiplied and accumulated to $\mathbf{C}^{(i,j)}$. Note that no movement of the data at the resulting matrix, $\mathbf{C}$, is necessary. All the message passings are within multiplicands $\mathbf{A}$ and $\mathbf{B}$.

For a k-processor machine, it is evident that the above algorithm is divided into k_s stages. Each stage consists of communications and computations. Computations in all the stages are identical. Communication step in the first stage only involves broadcasting of **A** while rest of the stages involve both broadcasting of **A** and shifting of **B**. The number of message-passings of each broadcast from a processor to $(k_s - 1)$ processors is clearly $(k_s - 1)$. Each shifting passes one message. Therefore, the total number of communications is given by $[k_s(k_s - 1) + (k_s - 1)] = (k - 1)$. However, the sizes of the messages vary with respect to the sizes of matrices **A** and **B**. If multiplicands **A** and **B** are of sizes $N_1 \times N_2$ and $N_2 \times N_3$, respectively, then total communication cost can be written as:

$$\begin{aligned} t_{mesh} &= (k-1)t_s + [k_s(k_s-1)(N_1N_2/k) + (k_s-1)(N_2N_3/k)]\, t_e \\ &= (k-1)t_s + (k_s-1)(N_2/k)\,[k_sN_1 + N_3]\, t_e \end{aligned} \tag{6.81}$$

where t_s is the start-up time for a communication and t_e is the communication time for one element. From equation (6.81), it is clear that the communication complexity is in the order of $O(k)$. However, the size of multiplicand **A** has more pronouncing effect on communication cost than size of multiplicand **B** because of the underlying broadcasts. An effort to eliminate either the broadcasts of left multiplicand or the shifts in right multiplicand results two extreme cases that are presented in the following section.

6.3 Two Extremes of Broadcast-and-Shift Algorithm

When either row-division (a set of complete rows is allocated to each processor) or column-division (a set of complete columns is allocated to each processor) data-allocations are considered for matrices **A** and **B**, either broadcasting **A** or shifting **B** can be eliminated. Figures 6.7(a) and (b) present the block-diagrams for these cases

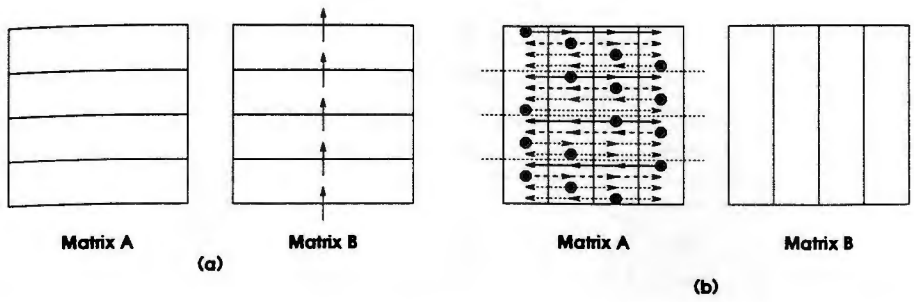


Figure 6.7: Broadcast-and-Shift Multiplication using 4-processor machine (a) for row-division with no broadcasts in **A** and (b) for column-division with no shifts in **B**.

showing the communication complexity, respectively. In case of row-division partitioning, matrix-vector multiplication would be efficient while in the case of column-division structure, vector-matrix multiplication could be efficient. This can be clearly seen in the following evaluation of communication costs for these extreme cases. On a k-processor machine with matrices **A** and **B** of sizes $N_1 \times N_2$ and $N_2 \times N_3$, respectively, broadcast-and-shift algorithm for these decompositions maps into k stages of shifts for row-division (see Figure 6.7(a)) or k stages of broadcasts for column-division (see Figure 6.7(b)) unlike k_s stages of broadcasts and shifts for mesh-division. Then, communication cost for row-division decomposition would be a result of shifts in **B**:

$$t_{row} = (k-1)t_s + (k_s - 1)(N_2/k)\left[(k_s+1)N_3\right]t_e \tag{6.82}$$

which is in the order of $O(k)$ while that for column-division decomposition can be derived from broadcasts in **A** as:

$$t_{col} = k(k-1)t_s + k(k-1)(N_1N_2/k^2)t_e \tag{6.83}$$

which is in the order of $O(k^2)$. However, when row-division partitioning is adopted for **A** and column-division decomposition is used for **B**, both the broadcast and shift communications are traded with communications of partial results of **C** that need to be accumulated. This case is studied in next section and compared to mesh-division broadcast-and-shift algorithm.

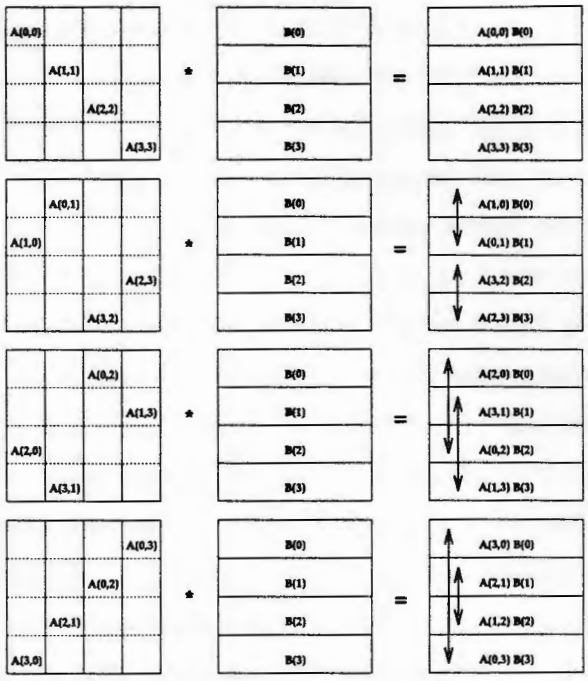


Figure 6.8: New Approach for Matrix Multiplication Algorithm on 4-processors

6.4 New Approach: Taking Advantage of Two Extremes

This section presents the unique domain decomposition of multiplicand matrices, **A** and **B**, that requires absolutely no messages being passed within multiplicands but involves only communication of partial results. Computational complexity is identical to that of broadcast-and-shift algorithm but communication complexity varies with sizes of multiplicand matrices. Moreover, storage requirements for new approach is less than that required in broadcast-and-shift algorithm since communication buffers are required for both the multiplicands in broadcast-and-shift algorithm while only a buffer as small as resulting matrix is required in our approach. Observing that dot product of two vectors result in one single element irrespective of length of the vectors, and considering the block transpose algorithm for row or column-division decompositions motivated us for this approach.

Consider two multiplicand matrices $\mathbf{A}$ and $\mathbf{B}$ of sizes $N_1 \times N_2$ and $N_2 \times N_3$, respectively, and $\mathbf{C} = \mathbf{AB}$. Let matrix $\mathbf{A}$ be decomposed using column-division while matrices $\mathbf{B}$ and $\mathbf{C}$ be decomposed with row-division. It is acknowledged that the data-partition considered here is not uniform for all the matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$. For embedding this algorithm into any computation would need necessary message-passings overhead to switch between data-partitions. Association of parts of each matrix to each node on a k-processor machine would result processor-i to contain $\mathbf{A}^{(i)}_{N_1 \times N_2/k}$, $\mathbf{B}^{(i)}_{N_2/k \times N_3}$, and $\mathbf{C}^{(i)}_{N_1/k \times N_3}$. Initially, it would seem that when each node contains parts of matrices $\mathbf{A}$ and $\mathbf{B}$ of sizes $N_1 \times N_2/k$ and $N_2/k \times N_3$, resulting matrix that obtained by their multiplication at each node would be of size $N_1 \times N_3$, which is as large as the entire resulting matrix that is supposed to be residing at all the k processors. However, using matrix transpose technique and further dividing the problem at each node, we can decompose algorithm into k successive compute, communicate, and accumulate stages, which would require a storage of size just $N_1/k \times N_3$. Total number of communications is exactly same as that required in broadcast-and-shift matrix multiplication algorithm but message-length varies as the sizes of matrices $\mathbf{A}$ and $\mathbf{B}$ depart from being square matrices. Moreover, restriction on broadcast-and-shift algorithm on number of processors to be square is no more applicable to this new approach.

Figure 6.8 demonstrates the new approach for matrix multiplication using column-division for left multiplicand $\mathbf{A}$ and row-division for right multiplicand $\mathbf{B}$ on a 4-processor machine. On a k-processor machine, suppose that processors are numbered as $P^{(0)} \ldots P^{(k-1)}$. Denote the part of matrix $\mathbf{A}$ that is allocated to processor i as $\mathbf{A}^{(i)}$. Similarly, denote the part of matrix $\mathbf{B}$ that is allocated to processor i as $\mathbf{B}^{(i)}$. Then, resulting matrix $\mathbf{C}$ can simply be expressed as

$$\mathbf{C} = \sum_{l=0}^{k-1} \mathbf{A}^{(l)}\mathbf{B}^{(l)}$$

If we break-up the computations into k stages, then we can form k communication stages followed by computation stages. This is accomplished by breaking-up associating matrix $\mathbf{A}$ at jth processor into k subblocks as $\mathbf{A}^{(i,j)}$ for $0 \leq j \leq (k-1)$. Then

k stages of computations are performed by multiplications: $\mathbf{A}^{(i,j)}\mathbf{B}^{(j)}$ at processor j for $i = 0$ to $(k-1)$. Then, $(k-1)$ stages of block-transpose algorithm can be interleaved with k stages of computations. Following equation explicitly shows the block-transpose algorithm where the horizontal lines represent node boundaries.

$$\left[\begin{array}{c} \mathbf{A}^{(0,0)}\mathbf{B}^{(0)} \\ \mathbf{A}^{(0,1)}\mathbf{B}^{(1)} \\ \vdots \\ \mathbf{A}^{(0,k-1)}\mathbf{B}^{(k-1)} \\ \hline \mathbf{A}^{(1,0)}\mathbf{B}^{(0)} \\ \mathbf{A}^{(1,1)}\mathbf{B}^{(1)} \\ \vdots \\ \mathbf{A}^{(1,k-1)}\mathbf{B}^{(k-1)} \\ \hline \vdots \\ \vdots \\ \mathbf{A}^{(k-1,k-1)}\mathbf{B}^{(k-1)} \end{array}\right] = \left[P(k^2,k) \otimes \mathbf{I}_{N_1N_3/k^2}\right] \left[\begin{array}{c} \mathbf{A}^{(0,0)}\mathbf{B}^{(0)} \\ \mathbf{A}^{(1,0)}\mathbf{B}^{(0)} \\ \vdots \\ \mathbf{A}^{(k-1,0)}\mathbf{B}^{(0)} \\ \hline \mathbf{A}^{(0,1)}\mathbf{B}^{(1)} \\ \mathbf{A}^{(1,1)}\mathbf{B}^{(1)} \\ \vdots \\ \mathbf{A}^{(k-1,1)}\mathbf{B}^{(1)} \\ \hline \vdots \\ \vdots \\ \mathbf{A}^{(k-1,k-1)}\mathbf{B}^{(k-1)} \end{array}\right] \tag{6.84}$$

Note that first (top-down) $(k-1)$ entries of right hand side matrix in equation (6.84) can be computed at processor-0 because subblocks $\mathbf{A}^{(j,0)}$ and $\mathbf{B}^{(0)}$, $0 \le j \le (k-1)$, are available at processor-0. Simultaneously, processor-1 computes products $\mathbf{A}^{(j,1)}\ \mathbf{B}^{(1)}$, $0 \le j \le (k-1)$, at processor-1, and so on. Then, after the transpose algorithm, computation can be completed by the following accumulation.

$$\left[\begin{array}{c} \mathbf{C}^{(0)} \\ \mathbf{C}^{(1)} \\ \vdots \\ \mathbf{C}^{(i)} \\ \vdots \\ \mathbf{C}^{(k-1)} \end{array}\right] = \left[\begin{array}{l} \sum_{j=0}^{(k-1)} \mathbf{A}^{(0,j)}\mathbf{B}^{(j)} \\ \sum_{j=0}^{(k-1)} \mathbf{A}^{(1,j)}\mathbf{B}^{(j)} \\ \vdots \\ \sum_{j=0}^{(k-1)} \mathbf{A}^{(i,j)}\mathbf{B}^{(j)} \\ \vdots \\ \sum_{j=0}^{(k-1)} \mathbf{A}^{(k-1,j)}\mathbf{B}^{(j)} \end{array}\right] \tag{6.85}$$

N_1 N_2 N_3	2-nodes	4-nodes	8-nodes	16-nodes
32 512 32	0.495	1.049	2.294	4.870
64 512 64	0.801	1.827	3.348	4.970
128 512 128	2.238	4.375	5.775	8.953
256 512 256	7.107	12.377	16.724	22.357
512 512 512	27.340	44.108	57.234	67.113

Table 6.8: Timing results for routing scheme in new matrix multiplication algorithm for 2, 4, 8 and 16-node partitions.

6.5 Performance Evaluation

We have seen in Section (6.2) that the communication overhead in broadcast-and-shift algorithm for running on k processors in equation (6.81). If we derive message-passing overhead inherent in the new approach in analogous fashion, then for k processors, implementation of multiplication of matrices of sizes $N_1 \times N_2$ and $N_2 \times N_3$ would involve a communication cost that can expressed as:

$$t_{new} = (k-1)\, t_s + (k-1)\, (N_1 N_3/k)\, t_e. \tag{6.86}$$

To compare the two algorithms, new approach performs better than broadcast-and-shift algorithm if

$$\begin{aligned} t_{mesh} &> t_{new} \\ k_s(k_s-1)(N_1N_2/k) &+ (k_s-1)(N_2N_3/k) > (k-1)\, (N_1N_3/k) \\ N_2 &> \frac{N_3}{\left[1+\left(\frac{N_3-N_1}{(k_s+1)N_1}\right)\right]} \end{aligned} \tag{6.87}$$

Note that both the algorithms have communication complexity that are in the order of $O(k)$ on a k-processor machine. Hence, the only difference arises from the sizes of the multiplicands and the resulting matrix. Clearly, above inequality says that if N_2 is larger than N_3, then messages in new approach will be shorter than that of communications in broadcast-and-shift algorithm. Tables 6.9, and 6.10 present results of actual implementations demonstrating the validity of the inequality (6.87).

N_1	N_2	N_3	B-S Algor.	New App.	Performance Improvement
128	128	32	11.811	5.384	119.35
128	128	64	9.769	7.589	28.73
128	128	128	10.313	9.290	11.02
256	128	32	12.108	7.538	60.63
512	128	32	15.429	9.330	65.37
1024	128	32	22.604	13.469	67.82
128	256	32	11.185	5.355	108.88
128	256	64	11.753	7.573	55.19
128	256	128	12.853	9.359	37.33
256	256	32	14.466	7.530	92.10
256	256	64	14.993	9.339	60.53
512	256	32	20.114	9.341	115.33
512	256	64	20.618	13.529	52.40
1024	256	32	37.661	13.518	178.59
128	512	32	15.005	5.273	184.58
128	512	64	16.127	7.511	114.71
128	512	128	18.651	9.336	99.78
128	512	256	22.296	13.496	65.21
256	512	32	20.647	7.571	172.70
256	512	64	21.557	9.360	130.31
256	512	128	24.351	13.468	80.81
512	512	32	32.073	9.333	243.65
512	512	64	32.874	13.487	143.74
1024	512	32	66.446	13.524	391.31
1024	512	64	54.994	23.743	131.63

Table 6.9: Timing results for routing schemes in matrix multiplication algorithms on Intel's Paragon with 16-processors.

Moreover, unlike broadcast-and-shift algorithm which is optimum for square number of processors [25], performance of new approach changes uniformly with increasing number of processors. This is demonstrated with our results in Table 6.5.

6.6 Conclusion

It is observed that variation of data-partitioning presents significant improvements in the performance of matrix multiplication algorithms for rectangular arrays. A clear analysis of an existing multiplication algorithm for distributed systems seeking minimum communication resulted in an efficient and new approach. Also, a quest to overcome the hurdles faced in memory requirements while developing this algorithm resulted in efficient utilizitation of the block-transpose algorithm seen in Chapter 3.

N_1	N_2	N_3	B-S Algor.	New App.	Performance Improvement
128	128	32	11.742	7.265	61.62
128	128	64	12.848	11.661	10.79
256	128	32	18.404	11.661	57.82
512	128	32	31.486	21.037	49.67
128	256	32	19.511	7.325	166.67
128	256	64	21.919	11.669	87.84
128	256	128	26.588	21.128	25.84
256	256	32	32.423	11.641	178.52
256	256	64	34.938	21.032	66.12
256	256	128	39.837	39.226	1.56
512	256	32	59.338	20.973	182.93
512	256	64	61.808	39.322	57.18
128	512	32	34.936	7.302	378.44
128	512	64	39.797	11.674	240.90
128	512	128	49.139	21.112	132.75
128	512	256	68.786	39.276	75.13
128	512	512	109.143	75.203	45.13
256	512	32	61.808	11.672	429.54
256	512	64	66.710	21.108	216.04
256	512	128	76.199	39.185	94.45
256	512	256	96.222	75.287	27.81
512	512	32	114.326	20.887	447.35
512	512	64	119.239	39.229	203.96
512	512	128	128.646	75.244	70.97

Table 6.10: Timing results for routing schemes in matrix multiplication algorithms on Touchstone Delta with 16-processors.

N_1	N_2	N_3	B-S Algor.	New App.	Performance Improvement
128	128	32	26.504	18.586	42.60
256	128	32	44.936	30.932	45.27
512	128	32	80.056	55.328	44.69
128	256	32	47.235	19.506	142.15
128	256	64	53.787	30.849	74.35
128	256	128	65.542	55.086	18.98
256	256	32	82.382	30.829	167.22
256	256	64	89.350	53.987	65.50
512	256	32	152.873	55.152	177.18
512	256	64	159.157	102.409	55.41
128	512	32	88.714	18.271	385.54
128	512	64	101.255	31.236	224.16
128	512	128	124.579	55.067	126.23
128	512	256	185.685	101.661	83.65
128	512	512	304.817	198.436	53.61
256	512	32	159.439	30.950	415.15
256	512	64	171.299	55.532	208.47
256	512	128	197.331	101.436	94.54
256	512	256	245.848	198.612	23.78
512	512	32	300.573	53.400	462.87
512	512	64	312.586	102.097	206.17
512	512	128	339.101	199.487	69.97

Table 6.11: Timing results for routing schemes in matrix multiplication algorithms on $iPSC/860$ with 16-processors.

Chapter 7

Conclusions and Future Research

It is well known that data-distribution in distributed memory multiprocessors is essential to achieving high performance of data parallel algorithms. The central feature of most implementations of these algorithms is the manner in which the expensive interprocessor communication is minimized.

We defined a set of expressions for partitioning data in multiprocessor environments using tensor products and stride permutations. Unlike the existing data-partitioning schemes, this representation can form a part of any algorithm that can be represented using tensor products and stride permutations. Hence, using the well established theorems in tensor algebra, one can easily manipulate the algorithm to clearly visualize the data migration stages.

The expressions defined for data-partitions have been used to demonstrate the representation of existing matrix transpose and two-dimensional FFT algorithms. For a practical application in which switching data-partitions was needed, manipulation of algorithms and derivations to interface among them proven to be successful by using our definitions for process of data-partition. This is seen in computing vorticity and stream functions via a partial differential equation solver that used wavelet-Galerkin method. Then, a variant for routing scheme in two and three-dimensional FFTs is derived that is applicable for today's large computers to solve huge data sizes. Finally, a data-allocation scheme is presented for matrix multiplication via transpose

algorithms for distributed systems. It is seen that such a distribution is efficient for rectangular matrices.

Tensor products and stride permutations have been extremely useful to transform an algebraic expression that is derived on paper into implementations on parallel machines. Algorithms derived and discussed in this dissertation were implemented on Intel's Paragon, Touchstone Delta, Gamma, and *iPSC/860*.

An immediate direction for future work based on this research is to introduce notation and develop relevant theorems required for parallel algorithms that cannot be represented using tensor products and stride permutations alone. Another research direction is to solve applications that have several computational modules which are efficient for distinct data-partition schemes to prove usefulness of our definitions for interfaces.

Appendix A

Tensor Product Representation of 3D-FFT

If we consider rows, columns, and depths as three axes of a three dimensional array, this appendix describes an algorithm in which domain decomposition strategy involves partitioning depth as shown in Figure A.1. Hence, each processor would have complete rows and complete columns but not complete depths.

Consider a three-dimensional array, $\mathbf{X}$, of size $L \times M \times N$ distributed onto k-processors as shown in Figure A.1. Then, similar to column-decomposition method, input and output data-shuffling matrices are identity matrices and hence do not affect the derivation. Then, 3D-DFT of $\mathbf{X}$ can be expressed using tensor notation

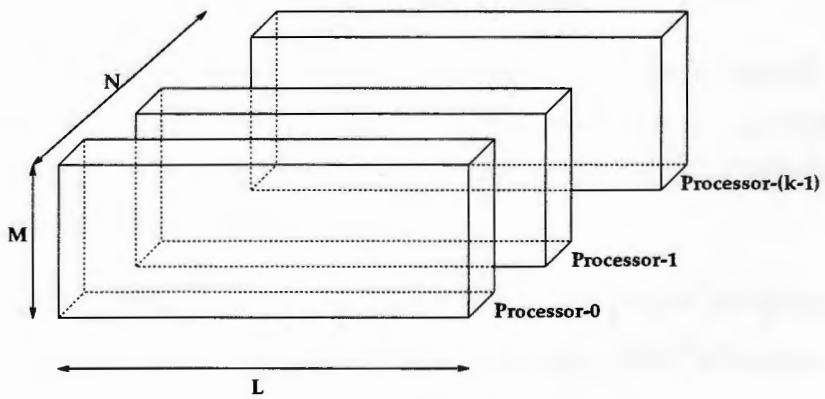


Figure A.1: Data-partitioning for Intel's 3D-FFT algorithm

as:

$$\mathbf{y} = [\mathbf{F}_N \otimes \mathbf{F}_M \otimes \mathbf{F}_N]\ \mathbf{x}, \tag{A.1}$$

where $\mathbf{x} = Vect_{LMN}(\mathbf{X})$ and $\mathbf{y} = Vect_{LMN}(\mathbf{Y})$. Then, implementation of equation (A.1) can be derived as follows:

$$[\mathbf{F}_N \otimes \mathbf{F}_M \otimes \mathbf{F}_L] = \underbrace{[\mathbf{F}_N \otimes \mathbf{I}_{ML}]}_{\mathcal{Z}_3}\underbrace{[\mathbf{I}_N \otimes \mathbf{F}_M \otimes \mathbf{I}_L]}_{\mathcal{Z}_2}\underbrace{[\mathbf{I}_{MN} \otimes \mathbf{F}_L]}_{\mathcal{Z}_1} \tag{A.2}$$

Then,

$$\mathcal{Z}_1 = \mathbf{I}_k \otimes \left[\mathbf{I}_{N/k} \otimes \mathbf{I}_M \otimes \mathbf{F}_L\right] \tag{A.3}$$

This is perfectly parallel with out any communications on p-processor machine.

$$\begin{aligned}
\mathcal{Z}_2 &= \mathbf{I}_N \otimes \mathbf{F}_M \otimes \mathbf{I}_L \\
&= \mathbf{I}_N \otimes [P(LM, M)\,(\mathbf{I}_L \otimes \mathbf{F}_M)\,P(LM, L)] \\
&= \mathbf{I}_k \otimes \left(\left[\mathbf{I}_{N/k} \otimes P(LM, M)\right]\left[\mathbf{I}_{LN/k} \otimes \mathbf{F}_M\right]\left[\mathbf{I}_{N/k} \otimes P(LM, L)\right]\right)
\end{aligned} \tag{A.4}$$

This consists of three perfectly parallel stages, first and third being vector-stride permutations that are inverse to one another, and second stage is LN/k number of one dimensional computations, each on a vector that is of length M.

$$\begin{aligned}
\mathcal{Z}_3 &= \mathbf{F}_N \otimes \mathbf{I}_{ML} \\
&= P(LMN, N)\,(\mathbf{I}_{ML} \otimes \mathbf{F}_N)\,P(LMN, ML)
\end{aligned} \tag{A.5}$$

Clearly, first and third stages consist communications, while second stage can be rewritten as $\left[\mathbf{I}_k \otimes \left(\mathbf{I}_{ML/k} \otimes \mathbf{F}_N\right)\right]$ implying that each of the k processors perform ML/k number of N-point FFTs. Communications in first and third stages can be revealed by further decomposition as:

$$\begin{aligned}
P(LMN, ML) &= [\mathbf{I}_k \otimes P(LMN/k, LM/k)]\left[P(kN, k) \otimes \mathbf{I}_{LM/k}\right] \\
&= [\mathbf{I}_k \otimes P(LMN/k, LM/k)]\left[P(k^2, k) \otimes \mathbf{I}_{LMN/k^2}\right] \\
&\quad \left[\mathbf{I}_k \otimes P(N, k) \otimes \mathbf{I}_{LM/k}\right]
\end{aligned} \tag{A.6}$$

Hence,

$$\begin{aligned} P(LMN,N) &= [P(LMN,ML)]^{-1} \\ &= \left[\mathbf{I}_k \otimes P(N,N/k) \otimes \mathbf{I}_{LM/k}\right]\left[P(k^2,k) \otimes \mathbf{I}_{LMN/k^2}\right] \\ &\quad [\mathbf{I}_k \otimes P(LMN/k,N)] \end{aligned} \tag{A.7}$$

Hence, equations (A.6) and (A.7) have similar structure except the local permutations stages are reversed. In both equations, second stages represent global communication that are seen before in transpose algorithms row or mesh-division decompositions. Hence, putting together all the above derivations, we can represent tensor notation of 3D-FFT algorithm on a k-processor machine as follows.

$$\begin{aligned} \mathbf{F}_N \otimes \mathbf{F}_M \otimes \mathbf{F}_L &= \left[\mathbf{I}_k \otimes P(N,N/k) \otimes \mathbf{I}_{LM/k}\right] \\ &\quad \left[P(k^2,k) \otimes \mathbf{I}_{LMN/k^2}\right] \\ &\quad [\mathbf{I}_k \otimes P(LMN/k,N)] \\ &\quad \left[\mathbf{I}_k \otimes \mathbf{I}_{LM/k} \otimes \mathbf{F}_N\right] \\ &\quad [\mathbf{I}_k \otimes P(LMN/k,LM/k)] \\ &\quad \left[P(k^2,k) \otimes \mathbf{I}_{LMN/k^2}\right] \\ &\quad \left[\mathbf{I}_k \otimes P(N,k) \otimes \mathbf{I}_{LM/k}\right] \\ &\quad \left[\mathbf{I}_k \otimes \mathbf{I}_{N/k} \otimes P(LM,M)\right] \\ &\quad \left[\mathbf{I}_k \otimes \mathbf{I}_{LN/k} \otimes \mathbf{F}_M\right] \\ &\quad \left[\mathbf{I}_k \otimes \mathbf{I}_{N/k} \otimes P(LM,L)\right] \\ &\quad \left[\mathbf{I}_k \otimes \mathbf{I}_{MN/k} \otimes \mathbf{F}_L\right] \end{aligned} \tag{A.8}$$

The above representation involves only two stages of communications, each stage consisting a complexity of $(k-1)$ number of messages, each message being a length of LMN/k^2.

Appendix B

Three Dimensional FFT using New Approach

In chapter 5, we promised that tensor notation helps to extend the problem for higher dimensions easily. In section 5.2, clear and distinct stages for two-dimensional FFT algorithm are presented in two sets (1) Rearrange I through Rearrange V in equations (5.68)-(5.73), and (2) Rearrange VI through Rearrange X. These results are proved in section 5.2.1. In this section, we present tensor formulation of new approach without proof for the case of three-dimensional array similar to one presented in section 5.2 for two-dimensional array.

Consider a three-dimensional data of size $L \times M \times N$ being Fourier transformed on k processors where these processors are arranged in $k_l \times k_m \times k_n$ grid. Due to the distribution, segmentation stages before and after the communication stages in each dimension are required.

$$\begin{aligned}
\textbf{Segment} &: \mathbf{I}_k \otimes P((N/k_n)k_l, k_l) \otimes \mathbf{I}_{LM/k_l^2 k_m} \\
\textbf{Rearrange I} &: \mathbf{I}_{k_n k_m} \otimes P(k_l^2, k_l) \otimes \mathbf{I}_{LMN/k_l^2 k_m k_n} \\
\textbf{Rearrange II} &: \mathbf{I}_k \otimes P((M/k_m)(N/k_n), (M/k_m)/k_l) \otimes \mathbf{I}_{L/k_l} \\
\textbf{Compute I} &: \mathbf{I}_k \otimes \mathbf{I}_{MN/k} \otimes F(L) \\
\textbf{Rearrange III} &: \mathbf{I}_k \otimes P((M/k_m)(N/k_n), (N/k_n)k_l) \otimes \mathbf{I}_{L/k_l}
\end{aligned}$$

$$\begin{aligned}
\textbf{Rearrange IV} &: \mathbf{I}_{k_n k_m} \otimes P(k_l^2, k_l) \otimes \mathbf{I}_{LMN/k_l^2 k_m k_n} \\
\textbf{Segment} &: \mathbf{I}_k \otimes P((N/k_n)k_l, N/k_n) \otimes \mathbf{I}_{LM/k_l^2 k_m} \\
\textbf{Rearrange V} &: \mathbf{I}_k \otimes P(LMN/k, L/k_l)
\end{aligned}$$

We have seen that the second set of operations for two-dimensional case are obtained by interchanging k_1 and k_2 and N_1 and N_2. However, in three dimensional case, we would need three sets. The second set is obtained with substitutions: $M \leftarrow L$, $N \leftarrow M$, and $L \leftarrow N$; and $k_m \leftarrow k_l$, $k_n \leftarrow k_m$, and $k_l \leftarrow k_n$. Similarly, the third set is obtained with substitutions: $N \leftarrow L$, $L \leftarrow M$, and $M \leftarrow N$; and $k_n \leftarrow k_l$, $k_l \leftarrow k_m$, and $k_m \leftarrow k_n$. Note that each set involves two communication stages, one computation stage, and rest are local permutations.

References

[1] D. B. Skillicorn, "A Taxonomy for Computer Architectures," *IEEE Computer*, pp. 46–57, Nov. 1988.

[2] R. Duncan, "A Survey of Parallel Computer Architectures," *IEEE Computer*, pp. 5–16, Feb. 1990.

[3] M. Singhal and T. L. Casavant, "Distributed Computing Systems," *IEEE Computer*, pp. 12–15, Aug. 1991.

[4] D. A. Reed and D. C. Grunwald, "The Performance of Multicomputer Interconnection Networks," *IEEE Computer*, pp. 63–73, June 1987.

[5] L. N. Bhuyan, "An Analysis of Processor-Memory Interconnection Networks," *IEEE Transactions on Computers*, vol. C-34, pp. 279–283, Mar. 1985.

[6] L. N. Bhuyan and D. P. Agrawal, "Design and Performance of Generalized Interconnection Networks," *IEEE Transactions on Computers*, vol. C-32, pp. 1081–1090, Dec. 1983.

[7] L. N. Bhuyan, Q. Yang, and D. P. Agrawal, "Performance of Multiprocessor Interconnection Networks," *IEEE Computer*, pp. 25–37, Feb. 1989.

[8] A. S. Tanenbaum, M. Frans Kaashoek, and H. E. Bal, "Parallel Programming Using Shared Objects and Broadcasting," *IEEE Computer*, pp. 10–19, Aug. 1992.

[9] A. H. Karp, "Programming for Parallelism," *IEEE Computer*, pp. 43–57, May 1987.

[10] W. J. Dally and C. L. Seitz, "The Torus Routing Chip," *Journal of Parallel and Distributed Computing*, vol. 1, pp. 187–196, 1986.

[11] W. J. Dally and C. L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Transactions on Computers*, vol. C-36, pp. 547–553, May 1987.

[12] L. M. Ni and P. K. McKinley, "A Survey of Wormhole Routing Techniques in Direct Networks," *IEEE Computer*, pp. 62–76, 1993.

[13] R. Tolimieri, M. An, and C. Lu, *Algorithms for Discrete Fourier Transform and Convolution.* Springer-Verlag Publishing Company, 1989.

[14] M. Davio, "Kronecker Products and Shuffle Algebra," *IEEE Transactions on Computers*, vol. C-30, pp. 116–125, 1981.

[15] H. Xu and L. M. Ni, "Optimizing Data Decomposition for Data Parallel Programs," in *International Conference on Parallel Processing*, pp. 225–232, 1994.

[16] J. Ramanujam and P. Sadayappan, "Compile-Time Techniques for Data Distribution in Distributed Memory Machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 472–482, Oct. 1991.

[17] M. Gupta and P. Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compiler on Multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 179–193, Mar. 1992.

[18] S. K. S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C. H. Huang, and P. Sadayappan, "On Compiling Array Expressions F Efficient Execution on Distributed-Memory Machines," in *Proceedings of International Conference on Parallel Processing*, pp. 301–305, 1993.

[19] S. D. Kaushik, C.-H. Huang, R. W. Johnson, and P. Sadayappan, "An Approach to Communication-Efficient Data Redistribution," in *Supercomputing 94*, pp. 364–373, 1994.

[20] J. Li and M. Chen, "The Data Alignment Phase in Compiling Programs for Distributed-Memory Machines," *Journal of Parallel and Distributed Computing*, vol. 13, pp. 213–221, Oct. 1991.

[21] K. Knob, J. D. Lukas, and G. L. Steel, "Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines," *Journal of Parallel and Distributed Computing*, vol. 3, pp. 102–118, Feb. 1990.

[22] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S. H. Teng, "Automatic Array Alignment in Data Parallel Algorithms," in *Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pp. 16–28, Jan. 1993.

[23] J. M. Anderson and M. S. Lam, "Global Optimization for Parallelism and Locality on Scalable Parallel Machines," in *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pp. 112–125, June 1993.

[24] M. An, I. Gertner, M. Rofheart, and R. Tolimieri, "Discrete Fast Fourier Transform Algorithms: A Tutorial Survey," *Advances in Electronics and Electron Physics*, vol. 80, 1991.

[25] G. Fox, A. J. C. Hey, and S. Otto, "Matrix Algorithms on the Hypercube I: Matrix Multiplication," *Parallel Computing*, vol. 4, pp. 17–31, 1987.

[26] Z. Qian and J. Weiss, "Wavelets and The Numerical Solution of Partial Differential Equations," *Journal of Computational Physics*, vol. 106, pp. 155–175, 1993.

[27] N. Anupindi, M. An, and Q. Yang, "Formulating Data Partition and Migration in Distributed Memory Multiprocessors," *submitted to Journal of Parallel and Distributed Systems*, 1994.

[28] A. Averbuch, E. Gabber, B. Gordissky, and Y. Medan, "A Parallel FFT on an MIMD Machine," in *Proceedings of International Conference on Parallel Processing*, pp. III–63–70, Aug. 1989.

[29] S. Horiguchi and T. Nakada, "Experimental Performance Evaluation of Parallel FFT on a Multiprocessor Workstation," in *Proceedings of International Conference on Parallel Processing*, pp. III–97–101, Aug. 1990.

[30] A. Norton and A. J. Silberger, "Parallelization and Performance Analysis of The Cooley-Tukey FFT Algorithm for Shared-Memory Architectures," *IEEE Transactions on Computers*, vol. C-36, pp. 581–591, May 1987.

[31] C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, "Firefly: A Multiprocessor Workstation," *IEEE Transactions on Computers*, vol. C-37, pp. 909–920, 1988.

[32] S. L. Johnson, M. Jacquemin, and C. T. Ho, "High Radix FFT on Boolean Cube Networks," Technical Report NA89-7, Thinking Machines Corporation, 1989.

[33] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures," *IEEE Transactions on Circuits, Systems, and Signal Processing*, vol. 9, pp. 449–500, 1990.

[34] P. N. Swarztrauber, "Multiprocessor FFTs," *Parallel Computing*, vol. 5, pp. 197–210, 1987.

[35] R. Susann, ed., *Parallel Programming.* McGraw-Hill Inc., 1991.

[36] A. Borodin, J. V. Z. Gathen, and J. Hopcroft, "Fast Parallel Matrix and GCD Computations," *Information and Control*, vol. 52, pp. 241–256, 1982.

[37] V. Strassen, "Gaussian Elimination is Not Optimal," *Numerical Mathematics*, vol. 13, pp. 354–356, 1969.

[38] C.-H. Huang, J. R. Johnson, and R. W. Johnson, "A Tensor Product Formulation of Strassen's Matrix Multiplication Algorithm," *Applied Mathematics Letters*, vol. 3, pp. 67–71, 1990.

[39] C.-H. Huang, J. R. Johnson, and R. W. Johnson, "Generating Parallel Programs from Tensor Product Formulas: A Case Study of Strassen's Matrix Multiplication algorithm," in *Proceedings of International Conference on Parallel Processing*, pp. 104–108, 1992.

[40] W. M. Gentleman, "Some Complexity Results for Matrix Computations on Parallel Processors," *Journal of ACM*, vol. 25, pp. 112–115, Jan. 1978.

[41] R. W. Numrich, ed., *Supercomputer Applications*, ch. A Vectorized Matrix-Vector Multiply and Overlapping Block Iterative Method by Linda J. Hayes. New York: Plenum Press, 1984.

[42] E. Dekel, D. Nassimi, and S. Sahni, "Parallel Matrix and Graph Algorithms," *SIAM Journal of Computing*, vol. 10, Nov. 1981.

[43] T. Agerwala and B. Lint, "Communication in Parallel Algorithms for Boolean Matrix Multiplication," in *Proceedings of International Conference on Parallel Processing*, pp. 146–153, 1978.

[44] R. P. Brent, "The Parallel Evaluation of General Arithmetic Expressions," *Journal of ACM*, vol. 21, pp. 201–206, 1974.

[45] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors, Volume I: General Techniques and Regular Problems.* Englewood Cliffs, New Jersy: Prentice Hall, 1988.

[46] S. L. Johnsson, "Minimizing the Communication Time for Matrix Multiplication on Multiprocessors," *Parallel Computing*, vol. 19, pp. 1237–1257, 1993.

[47] N. Anupindi, Q. Yang, and M. An, "Parallel Matrix Multiplication for Rectangular Arrays," *submitted to International Conference on Parallel Processing-95*, 1994.

[48] N. Anupindi, M. An, G. Kechriotis, and Q. Yang, "Generation of Distributed 2D-FFT Program using Tensor Algebra," *Submitted to 9th International Symposium on Parallel Processing-94*, 1994.

[49] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "IBM system/360 model 91: Machine philosophy and instruction handling," *IBM Journal of Research and Development*, pp. 8–24, Jan. 1967.

[50] M. C. Pease, "An adaptation of the fast Fourier transform for parallel processing," *Journal of ACM*, vol. 15, pp. 252–264, Apr. 1968.

[51] Z. Qian and J. Weiss, "Wavelets and The Numerical Solution of Boundary Value Problems," *Applied Mathematics Letters*, vol. 6, pp. 47–52, 1993.

[52] J. Weiss, "Wavelets and The Study of Two-Dimensional Turbulence," Technical Report AD910628, Aware Inc., One Memorial Dr., Cambridge, MA 02142-1301, 1992. and the Proceedings of French-USA Workshop on *Wavelets and Turbulence*, Princeton University, June 1991, Ed. Y. Maday, Springer-Verlag.

[53] K. Hwang, *Advanced Computer Architecture.* McGraw-Hill Book Company, Inc., 1993.

[54] N. Anupindi, M. An, J. W. Cooley, and Q. Yang, "A New and Efficient FFT Algorithm for Distributed Memory Systems," *to appear in International Conference on Parallel and Distributed Systems-94*, 1994.

[55] H. T. Kung and C. E. Leiserson, *Introduction to VLSI Systems*, ch. Section 8.3 by C. Mead and L. Conway. Reading, MA: Addison-Wesley, 1980.

[56] A. Sameh, "Numerical Algorithms on The Cedar Systems," in *Second SIAM conference on Vector and Parallel Processing in Scientific Computing*, (Virginia), 20^{th} November 1985.

[57] L. Johnsson and C.-T. Ho, "Matrix Multiplication on Boolean Cubes using Generic Communication Primities," in *Proceedings of the ARO workshop on Parallel Processing and Medium-Scale Multiprocessors*, 1986.

Bibliography

Agerwala, T. and Lint, B., "Communication in Parallel Algorithms for Boolean Matrix Multiplication," in *Proceedings of International Conference on Parallel Processing*, pp. 146–153, 1978.

An, M., Gertner, I., Rofheart, M., and Tolimieri, R., "Discrete Fast Fourier Transform Algorithms: A Tutorial Survey," *Advances in Electronics and Electron Physics*, vol. 80, 1991.

Anderson, D. W., Sparacio, F. J., and Tomasulo, R. M., "IBM system/360 model 91: Machine philosophy and instruction handling," *IBM Journal of Research and Development*, pp. 8–24, Jan. 1967.

Anderson, J. M. and Lam, M. S., "Global Optimization for Parallelism and Locality on Scalable Parallel Machines," in *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pp. 112–125, June 1993.

Anupindi, N., An, M., Cooley, J. W., and Yang, Q., "A New and Efficient FFT Algorithm for Distributed Memory Systems," *to appear in International Conference on Parallel and Distributed Systems-94*, 1994.

Anupindi, N., An, M., Kechriotis, G., and Yang, Q., "Generation of Distributed 2D-FFT Program using Tensor Algebra," *Submitted to 9th International Symposium on Parallel Processing-94*, 1994.

Anupindi, N., An, M., and Yang, Q., "Formulating Data Partition and Migration in Distributed Memory Multiprocessors," *submitted to Journal of Parallel and Distributed Systems*, 1994.

Anupindi, N., Yang, Q., and An, M., "Parallel Matrix Multiplication for Rectangular Arrays," *submitted to International Conference on Parallel Processing-95*, 1994.

Averbuch, A., Gabber, E., Gordissky, B., and Medan, Y., "A Parallel FFT on an MIMD Machine," in *Proceedings of International Conference on Parallel Processing*, pp. III–63–70, Aug. 1989.

Bhuyan, L. N., "An Analysis of Processor-Memory Interconnection Networks," *IEEE Transactions on Computers*, vol. C-34, pp. 279–283, Mar. 1985.

Bhuyan, L. N. and Agrawal, D. P., "Design and Performance of Generalized Interconnection Networks," *IEEE Transactions on Computers*, vol. C-32, pp. 1081–1090, Dec. 1983.

Bhuyan, L. N., Yang, Q., and Agrawal, D. P., "Performance of Multiprocessor Interconnection Networks," *IEEE Computer*, pp. 25–37, Feb. 1989.

Borodin, A., Gathen, J. V. Z., and Hopcroft, J., "Fast Parallel Matrix and GCD Computations," *Information and Control*, vol. 52, pp. 241–256, 1982.

Brent, R. P., "The Parallel Evaluation of General Arithmetic Expressions," *Journal of ACM*, vol. 21, pp. 201–206, 1974.

Chatterjee, S., Gilbert, J. R., Schreiber, R., and Teng, S. H., "Automatic Array Alignment in Data Parallel Algorithms," in *Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pp. 16–28, Jan. 1993.

Dally, W. J. and Seitz, C. L., "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Transactions on Computers*, vol. C-36, pp. 547–553, May 1987.

Dally, W. J. and Seitz, C. L., "The Torus Routing Chip," *Journal of Parallel and Distributed Computing*, vol. 1, pp. 187–196, 1986.

Davio, M., "Kronecker Products and Shuffle Algebra," *IEEE Transactions on Computers*, vol. C-30, pp. 116–125, 1981.

Dekel, E., Nassimi, D., and Sahni, S., "Parallel Matrix and Graph Algorithms," *SIAM Journal of Computing*, vol. 10, Nov. 1981.

Duncan, R., "A Survey of Parallel Computer Architectures," *IEEE Computer*, pp. 5–16, Feb. 1990.

Fox, G., Hey, A. J. C., and Otto, S., "Matrix Algorithms on the Hypercube I: Matrix Multiplication," *Parallel Computing*, vol. 4, pp. 17–31, 1987.

Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D., *Solving Problems on Concurrent Processors, Volume I: General Techniques and Regular Problems.* Englewood Cliffs, New Jersy: Prentice Hall, 1988.

Gentleman, W. M., "Some Complexity Results for Matrix Computations on Parallel Processors," *Journal of ACM*, vol. 25, pp. 112–115, Jan. 1978.

Gupta, M. and Banerjee, P., "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compiler on Multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 179–193, Mar. 1992.

Gupta, S. K. S., Kaushik, S. D., Mufti, S., Sharma, S., Huang, C. H., and Sadayappan, P., "On Compiling Array Expressions F Efficient Execution on Distributed-Memory Machines," in *Proceedings of International Conference on Parallel Processing*, pp. 301–305, 1993.

Horiguchi, S. and Nakada, T., "Experimental Performance Evaluation of Parallel FFT on a Multiprocessor Workstation," in *Proceedings of International Conference on Parallel Processing*, pp. III–97–101, Aug. 1990.

Huang, C.-H., Johnson, J. R., and Johnson, R. W., "Generating Parallel Programs from Tensor Product Formulas: A Case Study of Strassen's Matrix Multiplication algorithm," in *Proceedings of International Conference on Parallel Processing*, pp. 104–108, 1992.

Huang, C.-H., Johnson, J. R., and Johnson, R. W., "A Tensor Product Fromulation of Strassen's Matrix Multiplication Algorithm," *Applied Mathematics Letters*, vol. 3, pp. 67–71, 1990.

Hwang, K., *Advanced Computer Architecture.* McGraw-Hill Book Company, Inc., 1993.

Johnson, J. R., Johnson, R. W., Rodriguez, D., and Tolimieri, R., "A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures," *IEEE Transactions on Circuits, Systems, and Signal Processing*, vol. 9, pp. 449–500, 1990.

Johnson, S. L., Jacquemin, M., and Ho, C. T., "High Radix FFT on Boolean Cube Networks," Technical Report NA89-7, Thinking Machines Corporation, 1989.

Johnsson, L. and Ho, C.-T., "Matrix Multiplication on Boolean Cubes using Generic Communication Primities," in *Proceedings of the ARO workshop on Parallel Processing and Medium-Scale Multiprocessors*, 1986.

Johnsson, S. L., "Minimizing the Communication Time for Matrix Multiplication on Multiprocessors," *Parallel Computing*, vol. 19, pp. 1237–1257, 1993.

Karp, A. H., "Programming for Parallelism," *IEEE Computer*, pp. 43–57, May 1987.

Kaushik, S. D., Huang, C.-H., Johnson, R. W., and Sadayappan, P., "An Approach to Communication-Efficient Data Redistribution," in *Supercomputing 94*, pp. 364–373, 1994.

Knob, K., Lukas, J. D., and Steel, G. L., "Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines," *Journal of Parallel and Distributed Computing*, vol. 3, pp. 102–118, Feb. 1990.

Kung, H. T. and Leiserson, C. E., *Introduction to VLSI Systems*, ch. Section 8.3 by C. Mead and L. Conway. Reading, MA: Addison-Wesley, 1980.

Li, J. and Chen, M., "The Data Alignment Phase in Compiling Programs for Distributed-Memory Machines," *Journal of Parallel and Distributed Computing*, vol. 13, pp. 213–221, Oct. 1991.

Ni, L. M. and McKinley, P. K., "A Survey of Wormhole Routing Techniques in Direct Networks," *IEEE Computer*, pp. 62–76, 1993.

Norton, A. and Silberger, A. J., "Parallelization and Performance Analysis of The Cooley-Tukey FFT Algorithm for Shared-Memory Architectures," *IEEE Transactions on Computers*, vol. C-36, pp. 581–591, May 1987.

Numrich, R. W., ed., *Supercomputer Applications*, ch. A Vectorized Matrix-Vector Multiply and Overlapping Block Iterative Method by Linda J. Hayes. New York: Plenum Press, 1984.

Pease, M. C., "An adaptation of the fast Fourier transform for parallel processing," *Journal of ACM*, vol. 15, pp. 252–264, Apr. 1968.

Qian, Z. and Weiss, J., "Wavelets and The Numerical Solution of Boundary Value Problems," *Applied Mathematics Letters*, vol. 6, pp. 47–52, 1993.

Qian, Z. and Weiss, J., "Wavelets and The Numerical Solution of Partial Differential Equations," *Journal of Computational Physics*, vol. 106, pp. 155–175, 1993.

Ramanujam, J. and Sadayappan, P., "Compile-Time Techniques for Data Distribution in Distributed Memory Machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 472–482, Oct. 1991.

Reed, D. A. and Grunwald, D. C., "The Performance of Multicomputer Interconnection Networks," *IEEE Computer*, pp. 63–73, June 1987.

Sameh, A., "Numerical Algorithms on The Cedar Systems," in *Second SIAM conference on Vector and Parallel Processing in Scientific Computing*, (Virginia), 20^{th} November 1985.

Singhal, M. and Casavant, T. L., "Distributed Computing Systems," *IEEE Computer*, pp. 12–15, Aug. 1991.

Skillicorn, D. B., "A Taxonomy for Computer Architectures," *IEEE Computer*, pp. 46–57, Nov. 1988.

Strassen, V., "Gaussian Elimination is Not Optimal," *Numerical Mathematics*, vol. 13, pp. 354–356, 1969.

Susann, R., ed., *Parallel Programming.* McGraw-Hill Inc., 1991.

Swarztrauber, P. N., "Multiprocessor FFTs," *Parallel Computing*, vol. 5, pp. 197–210, 1987.

Tanenbaum, A. S., Frans Kaashoek, M., and Bal, H. E., "Parallel Programming Using Shared Objects and Broadcasting," *IEEE Computer*, pp. 10–19, Aug. 1992.

Thacker, C. P., Stewart, L. C., and Satterthwaite, E. H., "Firefly: A Multiprocessor Workstation," *IEEE Transactions on Computers*, vol. C-37, pp. 909–920, 1988.

Tolimieri, R., An, M., and Lu, C., *Algorithms for Discrete Fourier Transform and Convolution.* Springer-Verlag Publishing Company, 1989.

Weiss, J., "Wavelets and The Study of Two-Dimensional Turbulence," Technical Report AD910628, Aware Inc., One Memorial Dr., Cambridge, MA 02142-1301, 1992. and the Proceedings of French-USA Workshop on *Wavelets and Turbulence*, Princeton University, June 1991, Ed. Y. Maday, Springer-Verlag.

Xu, H. and Ni, L. M., "Optimizing Data Decomposition for Data Parallel Programs," in *International Conference on Parallel Processing*, pp. 225–232, 1994.